VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Komprimace bitmapových indexů

Bitmap index compression

2014 Lukáš Žák

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Lukáš Žák**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Komprimace bitmapových indexů
Bitmap Index Compression

Zásady pro vypracování:

Při práci s velkými daty v databázových systémech je nutné počítat s velkými nároky na primární i sekundární paměť. Pro snížení těchto nároků lze použít komprimaci. Cílem této práce je nastudovat a implementovat vybranou metodu komprimace bitmapových indexů.

1. Nastudujte metody komprimace bitmapových indexů.
2. Zvolenou metodu komprimace naimplementujte.
3. Proveďte testování a porovnejte zvolené metody komprimace.

Seznam doporučené odborné literatury:

Podle pokynů vedoucího bakalářské práce.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Václav Bašniar**

Datum zadání: 01.09.2013

Datum odevzdání: 07.05.2014

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA Fakulta elektrotechniky a informatiky OSTRAVA

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 18. července 2014 .....

Rád bych poděkoval panu Ing. Václavu Bašniarovi za vedení mé práce, za jeho věcné připomínky, rady, názory, trpělivost a především za jeho čas a podporu, kterou mi poskytl při sepisování dané problematiky.

Abstrakt

Cílem bakalářské práce je objasnit téma bitmapových indexů v databázových systémech a možnosti jejich komprimace. V úvodu se práce zabývá objasněním základních pojmů v této oblasti, jako jsou index v databázových systémech a jejich využití, konkrétní přiblížení bitmapových indexů a obecné pojmy komprimace. Hlavní část práce se poté zabývá metodotami komprimace bitmapových indexů, jejich výhodami a nevýhodami, kdy jedna z metod je detailně popsána, možnostmi použití komprimovaných bitmapových indexů a jejich správou. Dále je rozebrána implementace dané komprimační metody a její následné výkonnostní testování a hodnocení.

Klíčová slova: Databázový index, bitmapový index, index, komprimace, komprimace bitmapových indexů

Abstract

The main goal of the thesis is to clarify bitmap indexes in database systems and options of their compression. The introduction of this thesis is dedicated to basic terms in this field of work like what is index in database systems and their usage, specific closer look on bitmap indexes and terms of compression. The main part of the thesis deals with bitmap index compression methods, its main advantages and disadvantages and one of the methods is described in detail, possibilities of usage of compressed bitmap index and their management. Next and last part is focused on our compression method and its performance testing and evaluation.

Keywords: Database index, bitmap index, index, compression, bitmap index compression

Seznam použitých zkratek a symbolů

ASCII	–	American Standard Code for Information Interchange
BBC	–	Byte-Aligned Bitamp Code
CPU	–	Central Processing Unit
FIFO	–	First in, First out
ISO	–	International Organization for Standardization
LIFO	–	Last in, First out
RLE	–	Run length encoding
SŘBD	–	Systém řízení báze dat
WAH	–	Word-Aligned Hybrid

Obsah

Seznam tabulek

Seznam výpisů zdrojového kódu

1 Úvod

V dnešní době, kdy jsou kladeny velké nároky na informace a jejich dostupnost se neobejdeme bez rozsáhlých databázových systémů. Tyto systémy mohou ukládat velké množství informací různého druhu a různého rozsahu, ale hlavním a společným kritériem je jejich dostupnost, a to v co nejrychlejším čase a za vynaložení co nejméně prostředků. Prostředky může být myšleno, jak strojový čas procesoru strávený vyhledáváním podle kritérií, která mohou být v některých případech velice komplexní a složitá, tak i paměťové nároky. V případě hlavní paměti se jedná o časově náročné přístupy na fyzické médium disku a načítání daných informací, pokud nejsou již zavedeny v sekundární paměti. Ta má na rozdíl od primární paměti, tvořené pevnými disky, velmi omezenou kapacitu, ale nesrovnatelně vyšší rychlost a tím pádem rychlejší přístup k informacím. Pro tyto omezení je třeba zavést mechanizmus, který by usnadnil vyhledávání v tabulkách, a tím zrychlit a zefektivnit přístup k uloženým informacím.

K tomu to účelu se používá indexování tabulek. Indexů existuje několik druhů a každý se hodí pro jinou aplikaci. Proto bychom si před nastavením indexů nad sloupci měli rozmyslet, jak bude ve výsledné aplikaci sloupec využíván, aby zbytečnou aplikací indexů nedošlo k omezení výkonu namísto jeho nárůstu. Také bychom měli volit správný druh indexů podle druhu indexovaných dat, jejich množství a kardinalitě.

My se budeme zabývat bitmapovými indexy a podíváme se na jejich obecnou strukturu. Jejich stručnou historii, a s tím spojenou implementaci v komerčních databázových systémech. Jelikož jako každý druh indexu, i bitmapové indexy potřebují v databázích určitý úložný prostor, a aby se udržel v přípustné míře byly vyvinuty metody komprimace. Cílem bude prozkoumat metodu komprimace a naimplementovat ji společně s odpovídající metodou k její dekomprimaci. K správnému určení funkčnosti budou nezbytná také testovací data, která k tomuto účelu budou vygenerována, a na těchto datech bude prováděno testování dané metody komprimace.

V testech bude hlavním cílem ověřit správnost implementace jak komprimační, tak i dekomprimační metody, zhodnotit výkonost a také nezbytné určení efektivnosti komprimace, a tedy výslednou úsporu místa vzniklou komprimací.

2 Základní pojmy

2.1 Index v DB systémech

2.1.1 Co je to index

Index je datová struktura (B-strom, bitmapa, hash tabulka), která slouží ke zvýšení efektivity přístupu k datům uložených v tabulkách. Struktura indexu obsahuje informace o rozložení hodnot indexovaného sloupce a podle těchto informací se při vyhledávání přímo přistupuje k žádaným řádkům tabulky.

2.1.2 K čemu slouží a kde je použít

Databázové indexy slouží hlavně ke zrychlení přístupu k vyhledávaným datům. Při ukládání dat do tabulek nebývají vkládané záznamy nijak tříděny a ukládají se v pořadí, v jakém byly vkládány. Při následném vyhledávání je poté nutno sekvenčním způsobem procházet jednotlivé záznamy, než nalezneme námi požadovaný, což může být v nejhorším případě až záznam poslední, a je tedy zbytečné a neúčelné prohledávat všechny předchozí. K tomu, aby nebylo třeba prohledávat všechny předchozí záznamy slouží právě indexy, a tudíž je vhodné je vytvářet v tabulkách, nad kterými se bude provádět množství vyhledávacích dotazů, a to jak nad sloupci podle kterých se bude v záznamech vyhledávat, tak i nad sloupci, které budou sloužit pro třídění.

2.1.3 Použití

Index v tabulce může být definován, jak nad jedním sloupcem, tak i nad několika sloupci jako složený index a nad jednotlivými tabulkami může být vytvořeno několik indexů. Je důležité si uvědomit, že každý vytvořený index zabere v paměti systému určité množství úložného prostoru navíc (samotnou velikost dat nijak neovlivňují, ty zůstávají stále stejná) a důsledkem toho by při použití většího počtu indexů, zvláště u rozsáhlých tabulek mohlo dojít, že samotné indexy by v paměti databázového systému (neboli systém řízení báze dat - SŘBD) mohly zabírat více místa, než samotné data příslušné tabulky, což může být problematický a nežádoucí jev. Měly by se tedy používat hlavně tam, kde jsou důležité operace typu select a je předpoklad, že data se nebudou ve velké míře měnit.

2.1.4 Výhody a nevýhody

Výhodou používání indexů v databázi a zároveň hlavním důvodem jejich používání, je značně rychlejší vyhledávání v uložených datech, kdy není potřeba procházet všechny záznamy a přistupuje se pouze k relevantním řádkům tabulky. Naopak při vkládání a editaci dat budou nároky vyšší, protože dochází nejen k samotnému ukládání změn nebo přidávání záznamů do tabulky, ale také k úpravě či vkládání do indexu. [1]

[1] Database index. In: Wikipedia: the free encyclopedia [online]. Aktualizováno 22. 4. 2014. [cit. 2014-04-25]. Dostupné z: http://en.wikipedia.org/wiki/Database_index

2.2 Bitmapové indexy

Idea používání bitmapových indexů ke zrychlení vyhodnocování podmínek selekce byla rozvíjena již někdy od začátku 70. let dvacátého století. Přitom se bitmapové indexy nikdy nestaly součástí většiny z používaných komerčních databázových systémů. Jako jedna z mála společností nabízí možnost bitmapového indexování ORACLE, která jej nabízí od roku 1995. Další hlavní zástupci velkých databázových systému jako Microsoft SQL Server, BD2 od IBM a Sybase Adaptive Server jej nenabízejí. Nicméně Microsoft SQL Server může bitmapové indexy využívat při hash join. Tato situace, kdy většina systému bitmapové indexy nepodporuje je způsobena zčásti tím, že neexistuje žádný definitivní návrh nebo definice pro bitmapové indexy. Hlavní výhodou použití bitmapových indexů je jejich rychlost a nenáročnost na procesor počítače (CPU) při používání bitmapových operací jako AND, OR, XOR, které mohou využívat přímé hardwarové podpory procesoru. [2]

Obvykle indexy v sobě obsahují seznam RID [3]pro všechny záznamy v tabulce a to tak, že každý index obsahuje jedinečnou hodnotu indexovaného atributu. U bitmapových indexů je každý seznam RID reprezentován jako bitový vektor (t.j bitmapa), kde délka jednotlivých bitmap je úměrná kardinalitě[4] indexované relace a i-tý bit každé bitmapy odpovídá i-tému záznamu indexované relace. Je to nejjednodušší druh bitmapového indexu nazývaný simple bitmap index. [5] [6] [7]

Příklad 2.1
Mějme relaci o 12 záznamech a indexovaný atribut A který nabývá hodnot $v \in [0, 9]$ jak je vidět v tabulce 1. Bitmapa je kódována tak, že i-tý záznam obsahuje hodnotu v tehdy a pouze tehdy, pokud i-tý bit v bitmapě asociované s hodnotou atributu A je nastaven na 1 a i-tý bit, ve všech ostatních bitmapách je nastaven na 0. To znamená, že jeden řádek bitmapy představuje RID příslušného řádu relace. Každý řádek je tvořen sekvencí bitů svou velikostí odpovídající kardinalitě atributu A, v našem případě obsahuje řádek posloupnost 10 bitů označených jako E^0–E^9, kdy jednotlivé bity odpovídají jednotlivým jedinečným hodnotám atributu A. V prvním řádku relace nabývá atribut A hodnoty 3, je tudíž jasné, že pro správné vytvoření indexu pro tento záznam je třeba nastavit bit E^3 na hodnotu 1 a všechny zbylé bity nastavit na hodnotu 0. Tímhle způsobem jsme zaindexovali první záznam relace. Stejným postupem bychom postupovali dále pro všechny záznamy a vytvořili konečnou bitmapu. ■

[2] O'Neil, E.; O'Neil, P.; Kesheng Wu, "Bitmap Index Design Choices and Their Performance Implications,"*Database Engineering and Applications Symposium*, 2007. IDEAS 2007. 11th International , vol., no., pp.72,84, 6-8 Sept. 2007.

[3] Row ID – identifikátor řádku

[4] Mohutnost množiny

[5] Bitmap index. In: Wikipedia: the free encyclopedia [online]. Aktualizováno 27. 9. 2013. [cit. 2013-10-13]. Dostupné z: http://en.wikipedia.org/wiki/Bitmap_index

[6] LIU, Ling a M ÖZSU. Indexing of Data Warehouses In: *Encyclopedia of database systems*. New York: Springer,2009, 1454-1457. ISBN 978-038-7496-160.

[7] LIU, Ling a M ÖZSU. Bitmap index In: *Encyclopedia of database systems*. New York: Springer,2009, 244-248. ISBN 978-038-7496-160.

	A	E^9	E^8	E^7	E^6	E^5	E^4	E^3	E^2	E^1	E^0
1	3	0	0	0	0	0	0	1	0	0	0
2	2	0	0	0	0	0	0	0	1	0	0
3	1	0	0	0	0	0	0	0	0	1	0
4	2	0	0	0	0	0	0	0	1	0	0
5	8	0	1	0	0	0	0	0	0	0	0
6	2	0	0	0	0	0	0	0	1	0	0
7	9	1	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	1
9	7	0	0	1	0	0	0	0	0	0	0
10	5	0	0	0	0	1	0	0	0	0	0
11	6	0	0	0	1	0	0	0	0	0	0
12	4	0	0	0	0	0	1	0	0	0	0

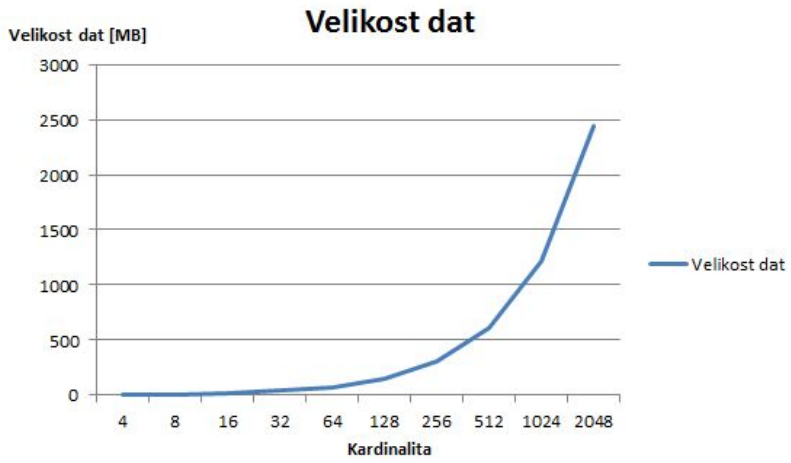
Tabulka 1: Simple bitmap

Nyní je již možné vyhledávání žádaných záznamů jen za pomoci jednoduchých bitových operací. Výhodou tohoto bitmapového indexu je jeho jednoduchost a kompaktnost datové struktury, kde zpracování zahrnuje pouze nenáročné logické operace. Další předností je efektivita kombinování několika dotazů dohromady pomocí bitových operací. Nicméně jelikož prostor potřebný k uchovávání simple bitmap indexu je přímo závislý na kardinalitě indexovaného atributu, tak efektivita simple bitmap indexu prudce klesá se zvyšováním kardinality indexovaného atributu. Na obrázku 1 vidíme graf zobrazující růst velikosti dat při zvyšující se kardinalitě pro 10 miliónů řádků.

2.2.1 Použití

Bitmapové indexy nám poskytují velice efektivní a výkonný způsob, jak provádět dotazy nad žádanými daty díky bitovým operacím podporovaných přímo hardwarem. Při použití simple bitmap indexu je vyhledávání prováděno jednoduše pomocí operaci OR, AND, nebo jejich kombinací podle zadaného dotazu, přímo nad bitovými vektory příslušných atributů vyskytujících se v dotazu. Po provedení všech potřebných operací nám vznikne výsledný bitový vektor, jehož jedničkové bity budou odpovídat námi kýženému výsledku dotazu a můžeme přistoupit ke korespondujícím záznamům. Pro větší názornost použijeme konkrétní příklad.

Příklad 2.2
Mějme tabulku 2 takovou, že bude obsahovat několik záznamů se dvěma indexovanými atributy, kterými jsou pozice a plat. Dále máme Sql dotaz 1 říkající nám, že máme vybrat všechna ID, pro která platí, že pozice="vedoucí" a plat="15,000" nebo plat="8,000". S tím, že nyní víme jaké hodnoty atributů budeme vybírat, můžeme určit konkrétní bitmapy, se kterými budeme provádět potřebné operace. První použitou bitmapou bude ta, které nám určuje, které řádky obsahují hodnotu *vedoucí* a takováto bitmapa bude vypadat



Obrázek 1: Velikost dat v závislosti na kardinalitě.

následovně: vedoucí: 01001010. Další potřebné bitmapy pro hodnoty atributu *plat* jsou 15,000: 11000000 a 8,000: 00010010.

Když známe všechny námi potřebné bitmapy, můžeme přikročit k samotnému vykonání dotazu. Nejprve je nutné provést bitový OR vektorů 11000000 a 00010010 pro získání vektoru, který nám bude říkat, které záznamy obsahují hodnotu atributu *plat* 15,000 nebo 8,000.

$$11000000 \vee 00010010 = 11010010$$

Po získání toho bitového vektoru nám již nic nebrání provedení bitového AND s vektorem, který nám udává *pozici*.

$$01001010 \wedge 11010010 = 01000010$$

Z výsledného bitového vektoru jasně vyplývá, že žádané záznamy v tabulce jsou druhý a sedmý.

```
SELECT ID
FROM pozice
WHERE pozice=vedouci AND (plat=15,000 OR plat=8,000)
```

Výpis 1: Sql dotaz

■

2.3 Komprimace

V informatice je pojmem komprimace považováno převádění (překódování) původních počítačových dat na data s menším datovým objemem, a to za podmínky, že informace v datech uchovávané zůstanou zachována. Hlavním cílem komprimace je zmenšení množství paměti potřebné k ukládání datových souborů nebo potřeba jejich přenosu přes sítě s omezenou kapacitou přenosu, a tudíž zkrácení doby pro přenos daného souboru.

ID	pozice	plat
1	ředitel	15,000
2	vedoucí	15,000
3	zaměstnanec	5,000
4	zaměstnanec	8,000
5	vedoucí	10,000
6	kuchař	5,000
7	vedoucí	8,000
8	zaměstnanec	5,000

Tabulka 2: Tabulka pozice

Komprimace může být jak ztrátová, tak bezztrátová. Nicméně pro správnou reprezentaci původních dat musí proběhnout dekomprimace, která stojí nějaký výpočetní čas i výkon.

Komprimaci dělíme na dva druhy a to:

1. Ztrátovou

   Algoritmy používané při ztrátové komprimaci analyzují data a odstraňují z nich bity, které vyhodnotí jako nepotřebné. Tím jsou některé informace nenávratně ztraceny, a díky tomu již není možné dosáhnout po dekompresi identického stavu dekomprimovaných dat s daty původními. Toho se využívá všude tam, kde takovéto ztráty určitého množství obsažených informací jsou přípustné. Schémata těchto komprimačních algoritmů bývají často založené na výzkumech lidského vnímání, hlavně sluchu a zraku, kdy člověk není schopen rozlišit ztrátu informací způsobenou komprimací. Hlavní výhodou těchto kompresních metod bývá dosažení vyššího komprimačního poměru, a tudíž větší úspora místa. Naopak nevýhodou je, že ztrátová komprimace je možná jen u určitých typů dat (převážně zvuková a obrazová), kdy ztráta informací do určité míry sníží kvalitu originálních dat, což je lidské vnímání schopno kompenzovat, nebo tam kde snížená kvalita nevadí a vyváží ji potřeba snížení potřebného místa k uložení. Z toho jasně vyplývá, že ztrátová komprese je nepřípustná u klasických počítačových dat, kdy i ztráta malého množství informaci při komprimaci by vedla k znehodnocení dat při následné dekomprimaci, a tím by se staly nepoužitelné.

2. Bezztrátovou

   Bezztrátová komprimace se vyznačuje tím, že při samotné komprimaci se neztrácejí žádné informace jako je tomu například u komprese ztrátové. Díky tomu to faktu je při dekomprimaci možné získat původní, nezměněná data a nedochází k jejich zkreslení nebo znehodnocení, jako tomu je u komprimace ztrátové. Proto je používaná u dat, u kterých by ztráta informací vedla k fatálnímu znehodnocení. Typickým příkladem takových dat mohou být textové soubory, kdy by ztráta mohla vyvolat nečitelnost nebo nesmyslnost textu po jeho dekomprimaci.

Míra komprimace daného souboru je dána výsledným kompresním poměrem a udává poměr mezi velikostí dat po komprimaci a jejich původní velikostí (tj. dat před kompri-

mací). Kompresní poměr je závislý, jak na druhu dat použitých ke komprimaci, tak na použitém kompresním algoritmu. Obecně lze říci, že kompresní poměr bývá vyšší u ztrátové komprese než u bezztrátové, ale její použití je limitováno druhem dat vhodných k takovému způsobu komprimace.[8] [9]

[8] Data compression. In: Wikipedia: the free encyclopedia [online]. Aktualizováno 9. 1. 2014 [cit. 2014-01-24]. Dostupné z: http://en.wikipedia.org/wiki/Data_compression

[9] KRČMÁŘ, Petr. Unixová komprese v praxi: Úvod. ROOT.CZ [online]. Aktualizováno 8. 3. 2003 [cit. 2013-11-5]. Dostupné z: http://www.root.cz/clanky/unixova-komprese-v-praxi-uvod/

3 Komprimace bitmapových indexů

Je fakt, že bitmapové indexy pro atributy, které mohou nabývat velkého množství různých hodnot by se brzy staly příliš velké a náročné na potřebné místo k jejich ukládání. Z tohoto důvodu se zavádí komprimace bitmapových indexů. Hlavním cílem komprimování indexů je zmenšení potřebného místa k uložení samotných indexů a zároveň zachování jejich výhod, konkrétně jejich výkonnost při provádění dotazů a to tak, že dotazy provádíme přímo nad komprimovanými indexy.

Řekněme, že máme tabulku T, která obsahuje n záznamů s indexovaným atributem A, který nabývá m různých hodnot. Z toho nám vyplývá, že výsledná bitmapa by byla nm velká. V případě, že hodnota m je relativně malá, nebude to ještě z pohledů spotřebovaného místa takový problém, ale s čím dál zvětšující se hodnotou m by se nám brzy mohlo stát, že velikost samotného indexu přeroste velikost indexovaných dat.

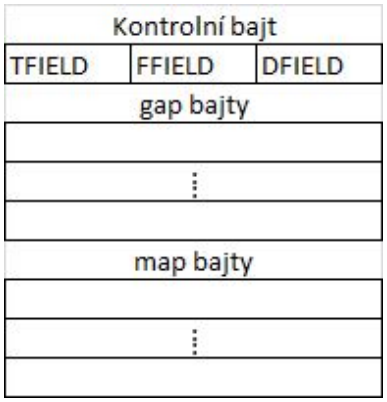
Na druhou stranu s rostoucím m budou v daném bitovém vektoru čím dál vzácnější výskyty logických jedniček. Přesněji pravděpodobnost výskytu logické 1 bude odpovídat $1/m$. S tímto poznatkem, že výskyt jedniček v bitovém vektoru bude řídký, můžeme jednotlivé bitové vektory vhodně zakódovat a tím dosáhnout toho, že budou v průměru zabírat méně než původních$n$ bitů. K tomu se využívá způsob obecně zvaný *run-length encoding*. V principu reprezentujeme sekvenci nul o velikosti i, následovanou jedničkou nějakou vhodnou binární reprezentací čísla i. Takto zakódované sekvence poté řadíme za sebe a tím nám vznikne výsledný zakódovaný bitový vektor.

Nicméně samotná binární reprezentace čísla i pro úspěšné zakódování nestačí. Při takovém jednoduchém zakódování bychom již nebyli schopni jednoznačně rozlišit jednotlivé zakódované sekvence. Například mějme bitový vektor 000101, který obsahuje 2 bitové sekvence, a to jednu délky 3 a druhou délky 1. Při převedení délky těchto sekvencí do binární podoby dostaneme 11 a 1 a výsledné zakódování původního vektoru by bylo 111. Z toho jde již vidět, že samotným zrcadlovým obrácením původního vektoru (101000) by výsledné zakódování bylo také 111. Díky vznikům těchto nejednoznačností není možné používat toto jednoduché schéma kvůli nemožnosti správného dekódování původního vektoru a jsme nuceni sáhnout po složitějších schématech.

Jako jedna z dalších možností komprimace bitmapových indexů se nabízí použití textového kompresního schématu, jako je například LZ77 využívaný v gzipu. Tato textová komprimační schémata se vyznačují svou efektivitou při zmenšování velikosti komprimovaného schématu. Jenže naším cílem není pouze zmenšit velikost bitmapového indexu, ale také zachovat možnost využití komprimovaného indexu k dotazování. U těchto druhů komprimace toho nejsme schopni docílit a jedinou možností, jak provádět operace nad takto komprimovaným indexem je celková dekomprimace. Ze samotného principu tedy vyplývá, že provádění databázových dotazů za pomoci takto komprimovaného indexu by bylo značně pomalé.

3.1 Byte-Aligned Bitamp Code

Další existující schéma již splňující požadavky na kompresi a možnost provádění databázových operací se nazývá *Byte-Aligned Bitamp Code*. Je to známé schéma, které dokáže



Obrázek 2: Struktura kódované sekvence.

zachovat efektivní provádění potřebných logických operací a zároveň dosahuje velmi dobrých výsledků při komprimaci. Díky tomu, že tato metoda vychází z principu RLE je značně jednoduší nežli metody textové, v našem případě zmíněné schéma LZ77. Také tedy využívá vyjádření délek jednotlivých sekvencí po sobě jdoucích stejných bitů vyjádřených jako hodnota bitu a délka sekvence. Schéma BBC nejprve danou bitmapu rozdělí do jednotlivých bajtů a tyto bajty sestaví do jednotlivých sekvencí. Tyto sekvence se dělí na takzvané gap, map a kontrolní bajt. Každý gap je tvořen bajty, jejichž bity obsahují buďto samé jedničky nebo naopak samé nuly. Bajty map jsou na rozdíl od gap tvořeny, jak jedničkami, tak nulami a vždy následují po gapu. V kontrolním bajtu poté ukládáme počty gapů, druh gapu a počet mapů. Tento kontrolní bit je složen ze tří polí TFIELD, FFIELD a DFIELD. Takto zpracovaný bitmapový index poté obsahuje sekvence kontrolního bajtu následovaný gap bajty a map bajty. Zpracování spočívá ve vyjádření samotných délek sekvencí. Nicméně nevyužívá jako délky počty bitů, ale díky samotnému uspořádání počet takto vzniklých bajtů. Takto vzniklé sekvence se poté skládají za sebe. Příklad můžeme vidět na obrázku 2.

Při komprimaci jsou pak prozkoumávány jednotlivé bity bitmapy. Pokud jednotlivé bajty po sobě jdoucí obsahují všechny bity stejné zakódují se jako gap. Poté se tento počet bajtů uloží do TFIELD pole v kontrolním bajtu. Pokud počet gapů je větší než 4 je tento počet zaznamenán ve gap bajtu. Pole FFIELD v kontrolním bajtu poté vyjadřuje hodnoty gapů. Jednička značí, že gapy obsahují samé jedničky, nula poté značí samé nuly. Pokud je bajt vyhodnocen jako map je tento bajt zapsán za předchozí map bajt nebo přímo za gap bajt. Po zapsání je zvýšena hodnota v poli DFIELD.

Při hodnotě v TFIELD 4-7 nastávají speciální případy. Při hodnotě 4 je počet gapů uložen v gap bajtu a následován 0-15 map bajty. Při hodnotě 5 je po kontrolním bajtu hned map bajt, který obsahuje pouze jediný rozdílný bit a jeho pozice je uložena v DFIELD. Počet gap bajtů je poté obsažen v FFIELD. Pro hodnotu 6 pak platí, že po gap bajtech následuje map bajt. Podle hodnoty FFIELD je poté určena hodnota všech bitů, jak gap, tak map s tím, že jediný bit v mapu je opačné hodnoty. Případ kdy TFIELD je hodnoty 7 je poté stejný jako v případě hodnoty 5 s tím rozdílem, že bity jsou negovány.

Obrázek 3: Dva druhy slov používaných u WAH.

Práce s takto komprimovaným indexem zahrnuje hlavně logické operace AND a OR. Při těchto operacích jsou potřebné části indexu dekomprimovány. S těmito dekomprimovanými částmi jsou následně provedeny požadované operace a je vytvořena sekvence reprezentující nám výsledek žádané operace nad zadanými bitmapami.
[10]

3.2 Word-Aligned Hybrid Code

Mezi komprimační schémata vycházející z RLE patří také *Word-Aligned Hybrid Code*. Tato metoda využívá základní velikost počítačové slovo. Nejčastější velikost slova bývá 32 bitů a dále 64 bitů. Pro znázornění budeme brát v potaz délku slova 32 bitů. V této metodě se využívají dva druhy slov pro vyjádření sekvencí jedniček a nul. A to slova typu fill a literal, které se rozlišují podle nejvyššího bitu slova viz. obrázek 3.

Náš nekomprimovaný bitmapový index tedy rozdělíme na jednotlivá slova délky 31 bitů (v případě, že zvolená délka slova je 32 bitů). Takovým to rozdělením nám vzniknou skupiny bitů, které mohou obsahovat buďto samé jedničky, nuly nebo jejich kombinaci. Pokud se je tato skupina bitů kombinací jedniček a nul vytvoříme z něj slovo typu literal. Takovémuto slovu nastavíme nejvyšší bit na nulu, která indikuje, že jde o slovo literal. Poté zbylé bity slova obsahují originální bity rozděleného indexu. Pokud rozdělená sekvence obsahuje samé nuly je nejvyšší bit slova nastavena na jedničku. Druhý nejvyšší bit poté je nastaven na nulu a značí, že zakódováváme sekvence obsahující samé nuly. V případě, že by jsme zakódovávali sekvence obsahující samé jedničky, bude druhý nejvyšší bit nastaven na jedničku. Zbylé bity dále obsahují počet sekvencí následujících za sebou, které všechny

[10] DIGITAL EQUIPMENT CORPORATION. *Byte aligned data compression* [online]. Vynálezce: Gennady Antoshenkov.. Přihl. 25.11.1994. MPT: G06T 9/00; H03M 7/46; H03M 007/00 ; H03M 007/46. Čís. patentu US5363098 A. The United States Patent and Trademark Office. [vid. 2.7.2014]. Dostupné na: http://patft.uspto.gov/netacgi/nph-Parser?Sect2=PTO1&Sect2=HITOFF&p=1&u=/netahtml/PTO/search-bool.html&r=1&f=G&l=50&d=PALL&RefSrch=yes&Query=PN/5363098

Obrázek 4: Struktura komprimované bitmapy.

Nekomprimovaná bitmapa	Komprimovaná bitmapa
0000 0000 0000 000	1000 0000 0000 0010
0000 0000 0000 000	0100 0011 0011 0011
100 0011 0011 0011	1100 0000 0000 0010
111 1111 1111 1111	aktivní slovo:
111 1111 1111 1111	1000 0011 1000 0000
100 0001 11	0000 0000 0000 1001

Tabulka 3: Ukázka komprimace. Index rozdelen do skupin po 15. Slova v komprimovaném indexu délky 16.

buď obsahují samé jedničky nebo nuly. To znamená, že k vyjádření počtu stejných za sebou jdoucích 31 bitových sekvencí obsahující samé jedničky nebo nuly máme k dispozici 30 bitů. Pokud by tato hodnota byla nedostačující nastaví se maximální hodnota a pokračuje se dále. Pokud poslední skupina bitů rozděleného indexu je velikosti 30 bitů a méně je ukládáno jako takzvané aktivní slovo (active word). Je ukládána jako slovo typu literal s tím rozdílem, že následující poslední slovo určuje počet využitých bitů v aktivním slově. Strukturu takto komprimované bitmapy lze videt na obrázku 4. V tabulce 3 pak můžeme vidět ukázku zakódování. V prvním a třetím řádku sloupce ”komprimovaná bitmapa”vidíme slova typu fill. První kóduje dvě sekvence nul, druhá poté dvě sekvence jedniček. V druhém řádku je ukázka slova literal. Na posledních řádcích vidíme aktivní slovo.

Pro srovnání, komprimační schéma WAH je jednoduší, a tudíž se dá předpokládat, že logické operace prováděné nad takto komprimovaným indexem budou rychlejší. To samé platí také pro komprimaci a dekomprimaci. V případě kódování relativně krátkých sekvencí vychází lépe BBC oproti WAH. To je dáno tím, že BBC pokaždé když narazí na krátkou sekvenci (tedy gap) zakóduje ji samostatně na výstup. Kdežto u komprimačního schématu WAH se použije typ slova literal, které bude obsahovat originální bity. Již z princip je pak jasné, že přistupovat k slovu literal, které obsahuje přímo dané bity bude

méně náročné nežli přistupovat k zakódované sekvenci v případě BBC. V případě WAH bude taky o něco horší kompresní poměr oproti BBC z důvodu, že pro vyjádření je třeba používat celá slova (např. délky 32 bitů).

Pro účely naší práce bylo zvoleno komprimační schéma run-length encoding popsáné níže. Bylo zvoleno z důvodu prosté reprezentace sekvencí nul a jejich zakódování a také proto, že z něj vycházejí výše popsané metody komprimace. Je tedy dobré pro reprezentaci základních principů, ze kterých se vychází u dalších metod komprimace bitmapových indexů, a dá se předpokládat i větší úspora místa v závislosti na tom, že výsledný komprimovaný index bude využívat pouze nezbytně nutný počet bitů. Vše je ovšem přímo závislé na vstupních datech, a tedy na samotném bitmapovém indexu.
[11] [12]

3.3 Run-length encoding

Při použití tohoto komprimačního schématu budeme schopni snížit potřebné množství bitů bitmapy, ale pouze za podmínky, že bitové sekvence, jak jsme si je představili výše, budou velmi dlouhé. Tím pádem velikost výsledné komprese bude velmi závislá na konkrétních komprimovaných datech a ne vždy dosáhneme ideálního kompresního poměru.

Nyní si popíšeme princip tohoto schématu a přiblížíme si jej na konkrétním příkladu pro větší názornost. Jako první krok musíme zjistit délku sekvence nul následovanou jedničkou a určit kolik je potřeba bitů k reprezentaci tohoto čísla i v binární soustavě. Toto zjištěné číslo j, které je přibližně $\log_2 i$, budeme dále reprezentovat jako sekvenci $j-1$ jedniček a následované jedinou nulou. Nyní jsme zakódovali počet následujících bitů, které již budou použity pro samotnou binární reprezentaci čísla i. Tímto postupem jsme zakódovali jednu sekvenci z našeho bitového vektoru. Opakováním postupu bychom postupně zakódovali celý náš bitový vektor a zřetězením všech výsledných kódů dostaneme již komprimovaný bitový vektor, u kterého jsme schopni jednoznačně rozlišit a určit hodnoty jednotlivých zakódovaných sekvencí a tudíž získat originální nezměněný bitový vektor.

Pro následné použití a správné dekódování využijeme právě číslo j, které jsme určili jako počet potřebných bitů binární reprezentace čísla i. Mějme naši zakódovanou sekvenci, kterou potřebujeme správně dekódovat a jsme na jejím začátku. Jelikož víme že, jsme na začátku sekvence, budeme bit po bitu číst jednotlivé hodnoty bitů, dokud nenarazíme na první nulu, k tomu abychom určili hodnotu čísla j. Toto číslo je počet bitů, které jsme prošli, dokud jsme nenarazili na první nulu včetně. Teď když již známe tuto hodnotu, tak víme, kolik přesně musíme projít následujících bitů pro správné načtení binární reprezentace hledaného čísla i. Díky tomuto postupu navíc přesně víme, kde nám

[11] THE REGENTS OF UNIVERSITY OF CALIFORNIA. *Word aligned bitmap compression method, data structure, and apparatus*[online]. Vynálezce: Kesheng Wu, Arie Shoshani, Ekow Otoo. Přihl. 14.12.2004. MPT: G06T 9/00; H03M 7/30; H03M 007/00. Čís. patentu US6831575 B2. The United States Patent and Trademark Office. [vid. 2.7.2014]. Dostupné na: http://patft.uspto.gov/netacgi/nph-Parser?Sect2=PTO1&Sect2=HITOFF&p=1&u=/netahtml/PTO/search-bool.html&r=1&f=G&l=50&d=PALL&RefSrch=yes&Query=PN/6831575

[12] WU, Kesheng; OTOO, Ekow J.; SHOSHANI, Arie. Optimizing bitmap indices with efficient compression. *ACM Transactions on Database Systems (TODS)*, 2006, 31.1: 1-38.

začíná další zakódovaná sekvence a můžeme opakováním postupu postupně dekódovat celý náš bitový vektor do jeho původní nekomprimované podoby.

Tímto postupem jsme samozřejmě zakódovali pouze jeden bitový vektor z naší bitmapy, a proto je potřeba jej aplikovat i na všechny zbývající bitové vektory, abychom dostali finální podoby celé komprimované bitmapy.

Příklad 3.1
Nyní si na příkladu názorně předvedeme výše popsaný postup komprimace jednotlivých bitových vektorů. Pro začátek si ukážeme, jak zakódovat sekvenci o délce 1 a 0, tudíž hodnoty $i = 1$ a $i = 0$. Pro obě tyto hodnoty platí, že pro jejich binární reprezentaci potřebujeme jeden bit takže $j = 1$ a podle našeho schématu je to tím pádem $j - 1$ jedniček následovaných jednou nulou. V tomto případě je počet jedniček nulový, a proto oba tyto kódy nebudou obsahovat na začátku žádné jedničky, ale pouze jednu nulu následovanou samotnou binární reprezentací čísla i. Takže zakódování pro $i = 0$ je 00 a $i = 1$ je 01.

Pro všechny ostatní případy již budeme postupovat standardně. Mějme číslo $i = 21$ potom $j = 5$, nyní víme, že pro binární reprezentaci i potřebujeme 5 bitů. Začátek kódování díky číslu j bude 11110 a dále bude následovat binární reprezentace čísla i, neboli 10101 a spojením získáme zakódování čísla 21, a to 1111010101. ■

Příklad 3.2
Pro náš další příklad si vezmeme již zakódovaný bitový vektor 11110101010011101101 a ukážeme si na něm správný postup dekódování jednotlivých sekvencí, ze kterých by bylo možné rekonstruovat původní bitový vektor. Pro správné rozklíčování, a tudíž pro určení délky jednotlivých sekvencí nul je potřeba nejprve číst po bitech náš vektor, dokud nenarazíme na jedničku, v našem případě je to bit pátý, takže $j = 5$. Již víme kolik následujících bitů musíme přečíst k získání hodnoty čísla i, a jelikož tyto bity jsou 10101 určíme, že první číslo je 21. Zbývající vektor pro dekódování nám nyní zbývá 0011101101. Hned první bit je 0, a tím pádem hned další bit reprezentuje samotné číslo, které je 0. Nyní jsme získali dvě sekvence, a to konkrétně 21, 0, a můžeme přistoupit k další části, která je 11101101. První nulový bit je na čtvrté pozici, a tím pádem poslední čtyři bity nám reprezentují poslední číslo sekvence, kdy 1101 = 13. Z konečného dekódování celé původní sekvence jsme zjistili, že sekvence nul původního nekomprimovaného vektoru jsou 21, 0 a 13 a z těchto poznatků rekonstruujeme nekomprimovaný vektor, konkrétně 00000000000000000000011000000000000001. ■

Již z principu této metody, kdy kódujeme sekvence nul následovaných jedničkou, dojdeme k závěru, že každý zpětně dekódovaný bitový vektor bude končit jedničkou a všechny případné nuly, které by se objevovaly dále nelze tímto způsobem získat. Jelikož se ale předpokládá, že počet záznamů v souboru, ke kterému se tento vektor vytvořil jako bitmapový index je nám znám, je tedy možné zbývající nuly, které nebyly dekódováním získány připojit na konec. Nicméně nuly nám v bitovém vektoru indikují, že záznam neobsahuje námi hledanou hodnotu. V tomto případě můžeme tedy chybějící nuly zcela zanedbat a nepotřebujeme ani znát celkový počet záznamů, ke kterým se daný index vztahuje a budeme stále schopni rozeznat, které záznamy obsahují hledanou hodnotu indexovaného atributu.

V předchozích příkladech jsme si ukázali, jak komprimovat i dekomprimovat jeden bitový vektor bitmapového indexu. Z těchto ukázek se může zdát, že výsledná komprese není nijak velká, a tudíž nebude velká ani konečná úspora paměti potřebné k uložení. Je to způsobeno tím, že tyto příklady jsou ilustrovány pro pochopení dané metody komprimace na velice malých vzorových datech. Pro přiblížení skutečné míry komprese a ušetřeného úložného prostoru si vezmeme pro příklad nejhorší případ, který by mohl nastat, a to že pro m záznamů by hodnota atributu n, pro který je vytvořen bitmapový index by byla unikátní, kdy $m = n$, a tudíž počet potřebných bitů a vytvoření bitmapového indexu pro tento atribut by byl mn. Všimněme si také, že kód pro délku sekvence i je $2 \log_2 i$ bitů a každý bitový vektor obsahuje pouze jednu jedničku. Díky tomu budeme mít vždy k zakódování pouze jednu sekvenci, která nemůže být delší než n, a proto je horní hranice počtu použitých bitů po zakódování $2 \log_2 n$. Pokud máme n bitových vektorů k zakódovaní v bitmapovém indexu (jelikož $n = m$), tak maximální počet potřebných bitů k zakódování takového indexu bude $2n \log_2 n$. Jak již bylo zmíněno nekomprimovaný index by zabíral n^2 bitů, ale dokud je $n > 4$, tak platí $2n \log_2 n < n2$ a při vzrůstající hodnotě n se bude komprimovaný index stávat stále výhodnější protože $2n \log_2 n$ bude stále více nabývat menších hodnot než n^2.

Další úspory místa potřebného k ukládání bitů při použití této metody můžeme docílit díky důsledku, že pro zakódování jednotlivých sekvencí používáme jen nezbytný počet bitů pro binární reprezentaci délky kódované sekvence nul. Při tomhle zjištění můžeme využít poznatku, že každá tato binární reprezentace bude začínat jedničkou (pro ilustraci zobrazeno v tabulce 4) a při kódování jednotlivých sekvenci můžeme tuto počáteční jedničku vynechat a pro binární reprezentaci délky sekvence používat právě jen $j - 1$ bitů. [13]Tudíž můžeme ušetřit jeden bit pro každou sekvenci a ušetřit tak další úložné místo. Toto se nám projeví hlavně při kódování dlouhého bitového vektoru, který bude obsahovat velké množství sekvencí, u kterých bude $j > 1$. Nicméně při zavedení tohoto mechanizmu, kdy nebudeme zapisovat první jedničku naší binární reprezentace, nesmíme zapomenout adekvátně upravit algoritmus pro dekódování, aby nedošlo k znehodnocení bitových vektorů při operacích s nimi prováděnými. [14] [15]

3.4 Operace nad komprimovaným bitovým vektorem

Po úspěšném zakódování našeho bitmapového indexu vyvstává zásadní otázka, jak s takovým komprimovaným indexem pracovat a provádět tak potřebné selekce konkrétních záznamů a bitové operace typu OR a AND. Nad komprimovaným indexem, a tudíž i na jednotlivých bitových vektorech není možné tyto operace provádět a nezbývá nám nic jiného než potřebný vektor nebo vektory v případě logických operací, správně dekódovat a všechny potřebné operace provádět až nad původním nekomprimovaným vektorem.

[13] Toto platí až na výjimku, kdy $j = 1$. V tomto případě nebudeme schopni rozlišit, jestli jsme takto zakódovali jedničku nebo nulu.

[14]ULLMAN, Jeffrey D.; GARCIA-MOLINA, Hector; WIDOM, Jennifer. *Database Systems: The Complete Book*. New Jersey: Prentice Hall, 2002, s. 702-704. ISBN 0-13-031995-3.

[15]ULLMAN, Jeffrey D.; GARCIA-MOLINA, Hector; WIDOM, Jennifer. *Database Systems: The Complete Book*. New Jersey: Prentice Hall, 2002, s. 704-706. ISBN 0-13-031995-3.

Dekadicky	Binárně
0	0
1	1
2	10
3	11
4	100
5	101
6	110
⋮	⋮
169	10101001

Tabulka 4: Ukázka binární řady

Jedním z možných způsobů, jak toho dosáhnout, je dekomprimovat celý potřebný vektor najednou a poté již pracovat s jeho nekomprimovanou formou. To je prospěšné hlavně v případě, kdy nepracujeme s více vektory najednou a nepožadujeme tím pádem s nimi provádět logické operace, ale pouze budeme požadovat například výpis všech záznamů odpovídajících určitému kritériu, ke kterému se vztahuje daný index.

Dalším možným způsobem, kterým můžeme postupovat je dekódovat jednotlivé sekvence postupně, a to díky tomuto komprimačnímu schématu. Vhodné bude zejména tam, kde budeme provádět logické operace nad více vektory (pro zjednodušení budeme dále popisovat operace mezi dvěma vektory) a již při samotném procesu dekódováním můžeme sestavovat vektor, který bude výsledkem konkrétní použité operace. Při použití tohoto postupu budeme jednotlivé vektory účastnící se příslušné operace dekódovat sekvenci po sekvenci a jsme tedy schopni okamžitě určit, na které pozici se objeví v konkrétním vektoru přísti jednička. V případě, že budeme provádět logický OR, tak můžeme okamžitě při zjištění pozice jedničky zapisovat také jedničku na příslušnou pozici výsledného vektoru, a to za předpokladu, že na danou pozici již jedničku nezapsal druhý vektor vstupující do této operace, v tom případě již není třeba tuto jedničku zapisovat znova. V případě logického AND je situace mírně odlišná a na výstup můžeme zapisovat pouze tehdy, když zjistíme, že oba vektory nám vyprodukují jedničku na stejné pozici. Musíme tedy předpokládat, že pokud nám jeden vektor určí jedničku na jedné určité pozici a druhý vektor na některé z pozic předchozích, nemůžeme zapsat na výstup jedničku, dokud nám druhý vektor neurčí, zda neobsahuje také jedničku na korespondující pozici. Opět si uvedeme konkrétní příklad pro přiblížení daného postupu.

Příklad 3.3
Pro ilustraci operací na zakódovaných vektorech jsme zvolili dva následující a pro lepší orientaci označil $v_1 = 101011010001$ a $v_2 = 1101110111$, na kterých si postup ukážeme. Již se nebudeme zabývat samotným dekódování, ale zaměříme se správné vyhodnocení logické operace nad zakódovanými vektory, a to konkrétně operace OR.

První délku sekvencí pro oba vektory jsme získali jednoduše a jsou to délky 2 a 7. Z toho můžeme usoudit, že první jednička vektoru v_1 bude na třetí pozici a první jednička

vektoru v_2 na pozici osmé.[16] Zapíšeme tedy dvě nuly na prvních dvou pozicích a jednu jedničku na pozici třetí našeho výstupního vektoru. Dále musíme dekódovat zbylé části našich vektorů, ale jelikož u v_1 se objevuje první jednička až na osmé pozici musíme prvně pokračovat v dekódování vektoru v_2, jelikož se může další jednička vyskytnou dříve než u v_1. Další sekvencí u v_2 je délky čtyři a tudíž následující jednička by zaujímala pozici 8, což je stejná pozice jako v případě v_1 a můžeme vygenerovat odpovídající počet nul a zapsat jedničku na osmou pozici výsledného vektoru. Následující délky sekvencí jsou tři pro v_1 a jedna pro v_2, takže zapíšeme jedničku na desátou pozici, a jelikož v_2 nám již nemůže vygenerovat žádnou další jedničku, zapíšeme rovnou jedničku i na pozici 12. Tímto jsme již dekódovali oba vektory a získali jsme výsledek námi žádané operace, který je 001000010101. Nyní jsme získali bitový vektor značící nám, které záznamy odpovídají hledaným kritériím.

Takto získaný vektor obsahuje 12 bitů indikující, kde se nacházejí hledané záznamy, nicméně předpokládejme, že počet záznamů v tabulce je patnáct. V tomto případě můžeme do konečného výsledku připsat zbývající nuly, ale stejně jako v příkladu výše to není nutné, jelikož by nám to stejně neindikovalo žádný další hledaný záznam. ■

17

[16] Předpokládejme, že indexy jednotlivých pozic číslujeme od jedničky

[17] ULLMAN, Jeffrey D.; GARCIA-MOLINA, Hector; WIDOM, Jennifer. *Database Systems: The Complete Book*. New Jersey: Prentice Hall, 2002, s. 706-707. ISBN 0-13-031995-3.

4 Údržba bitmapových indexů

Nyní když už víme, jak bitmapový index vypadá, jak se s ním pracuje a jak jej převést do jeho komprimované podoby pro ušetření paměti, vyvstávají další otázky vztahující se ke správě a údržbě takovýchto indexů. První důležitou otázkou je, jak vůbec přistoupit ke správnému bitovému vektoru, ke kterému je asociována námi požadovaná hodnota indexovaného atributu. Zatím jsme se zabývali pouze tím, jak index pro daný atribut vytvořit, ale nezabývali jsme se jak přistoupit ke správnému vektoru, který by nám dokázal odkázat na námi požadované záznamy, jelikož vytvořený index neobsahuje žádné informace o tom, ke kterým hodnotám se jednotlivé bitové vektory v něm obsažené vztahují.

Další navazující problém vzniká obdobně jako u toho prvního tím, že index neobsahuje žádné informace o tom, ke kterým hodnotám se vztahuje a zároveň také žádné informace o tom, které záznamy odpovídají těm námi hledaným po úspěšném vyhodnocení dotazu na tyto záznamy. Po vyhodnocení dostaneme prostý bitový vektor jehož jedničkové bity budou odpovídat hledaným záznamům, a tudíž musíme mít nějaký mechanizmus, jak z této informace správně určit dané záznamy. Při manipulaci s databázovými tabulkami se může a bude stávat, že se nám budou záznamy měnit, přidávat a odebírat, a to i pro záznamy nad kterými máme zavedeny bitmapové indexy. Musíme tedy také zajistit, aby se adekvátně měnili i příslušné vektory indexů, kterého se budou dané operace týkat a příslušně je upravit, aby stále odpovídaly reálnému stavu záznamů a bylo tedy možné s nimi nadále pracovat.

4.1 Určení bitového vektoru

Jak již bylo zmíněno bitmapový index je pouhá matice jednotlivých bitů nenesoucích žádné dodatečné informace, jak o sobě samém, tak o záznamech ke kterým se vztahuje. Je tedy nezbytné zavést nějakou techniku, tak abychom dokázali k jednotlivým bitovým vektorům přiřadit správnou hodnotu, které může daný atribut nabývat. Pro tuto potřebu si můžeme bitové vektory představit jako záznamy typu klíč-hodnota, kdy klíč v tomto případě bude odpovídající hodnota, které může nabývat daný atribut. A nyní můžeme tohoto využít a zavést sekundární index, který nás vždy odkáže na správný bitový vektor. Vhodným kandidátem na tento sekundární index se jeví B-strom jehož listy budou obsahovat dvojici klíč-ukazatel, kdy ukazatel nám odkáže na konkrétní hledaný vektor pro daný klíč. B-strom je vhodný i z toho důvodu, že jej lze jednoduše použít i pro dotazy na hodnoty v určitém rozsahu (range queries), ale lze samozřejmě použít i jiné typy indexu například hash tabulky a další.

4.2 Nalezení záznamu

Po nalezení bitových vektorů a vyhodnocení zadaného dotazu jsme dospěli k tomu, že jsme dostali vektor, který obsahuje jedničkový bit na i-té pozici a víme tedy, že takový záznam v tabulce existuje a musíme jej nějak získat a načíst. Opět si tuto situaci můžeme představit jako pár klíč-hodnota, kde klíč bude naše nalezená pozice i ve vektoru a opět bychom mohli mít zavedený sekundární index pro tyto potřeby získání záznamu.

Nicméně jelikož záznamy vkládané do tabulek nebývají nijak tříděny a nové záznamy bývají vkládány vždy nakonec tak to není potřeba. Za předpokladu, že čísla záznamů se po celou existenci tabulky nemění, i když byly některé záznamy smazány, můžeme pro získání záznamu použít primární index tabulky, případně rovnou primární klíč dané tabulky pokud používá jednoduché číslování záznamů a nejedná se o složený klíč nebo klíč vytvořený nějakými unikátními daty přímo daného záznamu. V těchto případech pak již můžeme jednoduše přistoupit k i-tému záznamů, který odpovídá svou pozicí pozici zjištěné z bitového vektoru vzniklého vyhodnocením dotazu.

4.3 Modifikace dat

Při modifikaci dat musíme přihlédnout ke dvěma problémům, které nastávají se změnou, či smazáním záznamů. Jako první, který souvisí s výše popsaným postupem na vyhledávání dat, musíme vzít v potaz, že se nám nesmí při modifikacích a mazání měnit číslování záznamů, aby bylo stále možné k nim správně přistupovat. A jako druhý problém nesmíme zapomenout při změnách záznamů patřičně upravit i již vytvořený bitmapový index, aby stále odpovídal realitě v tabulce a nevznikaly nám chybné výběry dat.

Jako první se podíváme na mazání dat. Nejjednodušší v tomto případě je mazaný záznam označit jako neaktivní a patřičným způsobem označit jeho bývalé úložné místo. Nesmíme také zapomenout na to odstranit jeho číslo tak, aby nemohlo být znovu použito, ale to nehrozí, pokud číslování záznamů pokračuje kontinuálně dále a nevyužívá čísla již smazaných záznamů. Dále je nutné správně pozměnit bitový index vztahující se k tomuto smazanému záznamu. Při mazání záznamu známe jeho hodnoty, kterých nabývaly jeho jednotlivé atributy a tudíž můžeme určit, který bitový vektor musíme upravit a z čísla mazaného záznamu také víme na které pozici. Nyní již stačí v daném vektoru a na dané pozici změnit původní jedničku, která indikovala indexovanou hodnotu atributu v mazaném záznamu na nulu.

Při vkládání záznamů je situace o něco jednodušší. Jelikož víme při vkládání další volné číslo pro označení záznamů, které tak přiřadíme novému záznamu a uložíme jej. Dalším nezbytným krokem je zjistit hodnotu atributu, který je indexován naším bitmapovým indexem a patřičně upravit všechny vektory indexu. Po zjištění hodnoty, které nabývá indexovaný atribut přistoupíme k odpovídajícímu vektoru indexu a modifikujeme jej tak, že přidáme na jeho konec jedničku, která nám od teď bude indikovat, že se hodnota korespondující s tímto vektorem objevuje v nově přidaném záznamu. Pro všechny ostatní vektory je modifikace podobná a to tak, že místo přidání jedničky nakonec vektoru přidáváme nulu.

Speciální případ může nastat tehdy, pokud přidáváme záznam s novou, ještě se v tabulce nevyskytující hodnotou atributu, pro který je vytvořen bitmapový index. Pokud tato situace nastane je nutné zanést tuto novou hodnotu. jak do bitmapového indexu, tak do sekundárního indexu odkazujícím nám na jednotlivé bitové vektory. Jako první musíme novou hodnotu promítnout do struktury bitmapového indexu, kde vytvoříme nový bitový vektor obsahující samé nuly (počtem odpovídající aktuální velikosti indexu), kromě poslední pozice, která bude obsahovat jedničku a indikovat právě vloženou hodnotu a samozřejmě všem ostatním vektorům vyskytujícím se v indexu přibude nula na

jejich konec. Další nutnou operací je zavést tuto novou hodnotu do sekundárního indexu, aby bylo možno správě určit nově přidaný bitový vektor podle hodnoty indexovaného atributu. [18]

[18] ULLMAN, Jeffrey D.; GARCIA-MOLINA, Hector; WIDOM, Jennifer. D*atabase Systems: The Complete Book*. New Jersey: Prentice Hall, 2002, s. 707-709. ISBN 0-13-031995-3.

5 Implementace

V následující kapitole se již budeme zabývat konkrétní implementací dané metody komprimace. Popíšeme si, jak algoritmus potřebný pro správnou kompresi, tak i potřebný dekompresní algoritmus, bez kterého by byly komprimované data bezcenná. Další nedílnou součástí tohoto programu je zvolit vhodný formát ukládání bitmapového indexu v paměti, aby bylo možné data číst a také vytvářet výslednou komprimovanou formu indexu, s tím nedílně souvisí i zvolení formy fyzického ukládání na pevný disk, tak aby nedocházelo k nesprávné interpretaci při čtení nebo zápisu, jak komprimovaného, tak i nekomprimovaného indexu. Další nutností je mít vstupní data reprezentující nám náš bitmapový index. Pro naše účely budou vygenerována testovací data o různé velikosti a hustotě rozložení jedniček v indexu.

Implementovaná metoda bude Run-lenght encodig, která byla teoreticky popsána v kapitole 2 a zde si předvedeme její implementaci v programovacím jazyce C++. Stručně charakterizujeme jazyk C++, který byl použit k implementaci a podíváme se obecně i na metodu RLE bez kontextu k bitmapovým indexům. Postupně projdeme důležité datové struktury použité v programu a vytvořené algoritmy pro generování vstupních dat, komprimaci i následnou dekomprimaci.

5.1 C++

C++ je široce rozšířeny programovací jazyk vyznačující se svou vysokou rychlostí a moderními prvky objektově orientovaného programování. Díky jeho rozšířenosti je pro něj dostupná spousta vývojových a ladících nástrojů, v našem případě Microsoft Visual Studio ve verzi 2012. Jakožto jeden z nejrozšířenějších jazyků s množstvím kompilátorů na různých platformách, je u programů napsaných v C++ velká přenositelnost mezi platformami s minimem potřebných změn kódu.

Je to kompilovaný jazyk překládaný přímo do strojového kód, a díky tomu se může řadit k jednomu z nejrychlejších jazyků, za předpokladu správné optimalizace. Další z jeho výhod je standardizace pod záštitou organizace ISO. Nedílnou součástí jazyka C++ je také jeho kompatibilita s jazykem C, vyznačující se tím, že programy a knihovny vytvořené v jazyce C budou s minimálními či žádnými úpravami fungovat i v C++. [19]

5.1.1 Historie

Historie programovacího jazyka C++ sahá až do roku 1979 a za jeho vývojem stál Bjarne Stroustrup. Jeho hlavním cílem bylo upravit stávající jazyk C, známý svou dobrou přenositelností mezi platformami i architekturami při zachování jeho rychlosti. Začal tedy vyvíjet projekt nazvaná „C with Classes“ obohacující jazyk C o třídy a dědičnost při zachování funkcionality jazyka C. První kompiler pro tento jazyk se stal Cfront překládající C with Classes do jazyka C.

V roce 1983 se tento jazyk přejmenoval již do dnešní podoby, a to C++ a byly přidány další nové a rozšiřující funkce. Nedlouho nato začaly vznikat první komerční produkty,

[19] A Brief Description. [online]. [cit. 2014-03-01]. Dostupné z: http://www.cplusplus.com/info/description/

nicméně jazyk jako takový stále nebyl standardizován. První mezinárodní standard vydala organizace ISO až v roce 1998 nazvaný C++ ISO/IEC 14882:1998 a odtud také pochází obecně užívaný název C++98. Další standard vyšel v roce 2003, který opravoval problémy předchozí verze. Zatím poslední verze standardu je z roku 2011 a nese označení ISO/IEC 14882:2011. [20]

5.2 RLE

V kapitole 2 jsme podrobně probrali komprimační metodu RLE ve vztahu k bitmapovým indexům, která je specifická svými vstupními daty, které jsou reprezentovány pouze jedničkami a nulami a také tím, že jsou kladeny nároky na výstupní komprimovaná data ve smyslu nutnosti rozlišit, dvě významově rozličné sekvence udávající nám počet bitů, které byly potřeba k zakódování originální sekvence a poté adekvátní počet bitů které představují binární vyjádření čísla reprezentující délku sekvence po sobě jdoucích nul následovaných jedničkou.

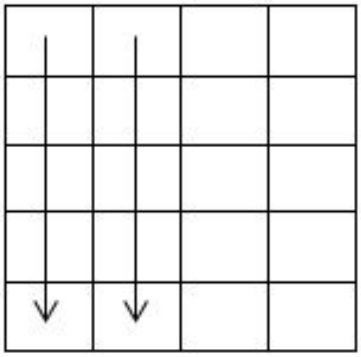
Nicméně námi použitá interpretace RLE je značně specifická pro danou řešenou úlohu. Obecně se RLE dá popsat takto: pokud se prvek p ve vstupních datech vyskytuje v určitém počtu n po sobě jdoucích, tak je možné tuto původní řadu prvků p nahradit párem složeným, jak z prvku samotného, tak z počtu jeho po sobě jdoucích opakování a výstupní data budou odpovídat páru pn. Této metody se nevyužívá pouze u bitmapových indexů, ale i tam kde se vyskytují souvislé opakování určitých dat, jako je například text nebo obrázky skládající se z pixelů.

Komprimace textu metodou RLE se může zdát intuitivní tak, že každý opakující se znak na vstupu převedeme na výstup v podobě znaku samotného následovaného číslem vyjadřujícím počet jeho opakování. Toto platí pouze do doby, než se nám ve vstupních datech spolu s textem začnou objevovat číslice, v tomto případě již nebudeme schopni při zpracování komprimované verze rozlišit, zdali se jedná skutečně o číslici nebo reprezentuje počet opakování předchozího znaku. Tomuto se dá předejít, použijeme-li před dvojicí znak, počet opakování speciální znak uvozující nám, že následující znaky jsou komprimovaný pár. Je také nutné si uvědomit možnost výskytu speciálního znaku použitého pro uvození v samotném textu a následného znehodnocení komprimovaného výstupu. Další z nepříjemností plynoucí z podstaty psaného textu je malý výskyt po sobě jdoucích, opakujících se znaků. V anglicky psaném textu je množství zdvojených znaků, nicméně výskyt ztrojených je vzácný a vezmeme-li v potaz český jazyk, výskyt zdvojených, natož ztrojených znaků je minimální. U psaného jazyka tedy nebude komprese účinná, dokonce se může výsledný text stát delší než původní díky výskytu uvozujícího znaku a ze znaků „nn“ by vzniklo „@n2“ [21]

Kromě komprimace textu se run length encoding jeví jako kandidát na komprimaci grafických dat, konkrétně digitálních obrázků. Obrázky skládající se z jednotlivých pixelů v stupních šedi nebo barevných jsou uloženy ve struktuře zvané bitmapa a principiálně jsou podobné bitmapovým indexům, kterými jsme se zabývali s tím rozdílem,

[20] History of C++. [online]. [cit. 2014-03-01]. Dostupné z: http://www.cplusplus.com/info/history/

[21] použití znaku @ je pouze ilustrativní a lze použít znak jiný nebo jeden z kontrolních znaků ASCII.



Obrázek 5: Směr procházení bitů.

že jednotlivé body (pixely) bitmapy jsou reprezentovány bity, jejichž hodnota odpovídá barvě příslušného pixelu. Komprese takových to obrázků spočívá na předpokladu, že pokud zvolíme libovolný pixel, tak jeho sousedé budou nabývat téže barvy. To je výhodné zejména u obrázků s nízkou členitostí a malou barevnou hloubkou, kde je větší šance na shodu sousedících pixelů. Při použití této metody na složitý a členitý obrázek například fotografii se může stát výstupní komprimovaný obrázek naopak velikostně větší než originál. [22]

5.3 Kód

Vytvořený program se skládá z několika funkcí zajišťujících generování potřebných dat, které slouží pro následnou komprimaci, samotnou komprimaci, jež společně s dekomprimací je hlavní složkou programu, a nakonec z funkcí podpůrných sloužící k zajištění zapisování a čtení dat nebo převody mezi číselnými soustavami.

Jako první je třeba vybrat vhodný datový typ, kterým budeme reprezentovat bitmapový index v paměti programu, a který bude také čten ze souboru a také do něj zapisován. Pro tyto účely byl jako vhodný datový typ zvolen unsigned char. Vybrán byl z důvodu, že se jedná o primitivní datový typ, který má od ostatních primitivních typů velikost pouze jednoho bajtu [23]a zmírní se tím velikost přesahu jednotlivých bitových vektorů indexu, u kterých nebude počet jejich bitů násobkem osmi. V případě bitmapového indexu se může jako vhodný datový typ jevit bool indikující nám přímo hodnotu daného bitu, který v případě jazyka C++ zabírá taktéž 1 bajt, nicméně by onen bajt ukládal informaci pouze o jednom bitu a ve výsledku by bitmapový index takto uložený v paměti zabíral 8 krát více místa, zatímco u unsigned char můžeme nastavovat jednotlivě všech 8 bitů a docílit tak téměř stejného počtu bitů potřebných k uložení jako obsahuje bitmapový index. Vybrané datové typy a jejich velikost jsou ilustrovány v tabulce 5.

[22]SALOMON, David. *Data Compression: the complete reference*. 4th ed. London: Springer, 2007, s. 23-31. ISBN 978-1-84628-602-5.

[23]Pojmem bajt budeme označovat jeden oktet tj. 8 bitů. Jelikož v dřívějších architekturách mohl být bajt složen i z jiného počtu bitů nežli dnešních osmi.

Typ	Velikost
bool	1 bajt
char	1 bajt
int	4 bajty
float	4 bajty
double	8 bajtů

Tabulka 5: Přehled vybraných datových typů a jejich velikost

Velikost unsigned charu samozřejmě není dostačující pro uložení celého indexu a je nutné zvolit takovou strukturu, která nám bude jednotlivé prvky spojovat do jednoho celku, nejlépe dvojrozměrné struktury pro intuitivní reprezentaci souvisejících prvků jednotlivých bitových vektorů indexu. Jako první kandidát na tuto strukturu můžeme vybrat dvojrozměrné pole skládající se z prvků typu unsigned char. Pole je vlastně souvislé vyhrazené místo v paměti pro prvky daného datového typu, ke kterým se přistupuje pomocí ukazatelů na tento typ. Bohužel v jazyce C++ je nutnost velikost pole definovat již při jeho vytváření tak, aby kompilátor již v době překladu věděl, jaká bude velikost daného pole. Nelze tedy pro velikost pole použít například proměnnou, ale pouze konkrétní číslo nebo konstantu. Z toho nám plyne omezení při komprimaci, kdy předem nevíme, jak dlouhé budou výsledné komprimované bitové vektory. Předejít tomuto stavu by bylo možné použitím dynamicky alokovaných polí a pro přidávání nových prvků vždy dané pole ručně realokovat a tím zvětšit jeho velikost. Abychom se vyhnuli ruční realokaci pro každý přidávaný prvek, C++ nabízí řadu standardních datových kontejnerů. Jsou to vlastně šablony zastřešující nám uložení a přístup k datům určitého typu. Tyto šablony jsou dostupné v Standard Template Library (STL) a označovány jako STL Containers, která je součástí standardu ISO a tato knihovna by měla být dostupná v každém překladači pro jazyk C++.

Nesmíme také zapomenout zmínit kontejner nepatřící do STL, ale zdající se pro danou aplikaci velice vhodný. Jedná se o bitset, který ukládá jednotlivé bity. Bitset je v podstatě pole prvků simulujících datový typ bool pro ukládání jedniček a nul, respektive true a false, s tím rozdílem, že pro tuto ukládanou hodnotu bude opravdu využit pouze jeden bit paměti. Je tedy paměťově optimalizovaný a díky tomu v paměti zabírá osm krát méně než nejmenší primitivní typ char. K jednotlivým bitům je pak možné přistupovat, obdobně jako u klasického pole, pomocí operátoru [ ]. Z důvodu absence datového typu, který by byl právě jeden bit, obsahuje bitset speciální referenční typ, který mu dovoluje manipulovat s jednotlivými uloženými bity. Je to vlastně proxy třída poskytující referenci, a tím tedy i přístup k jednotlivým bitům. Další z vlastností bitsetu je jeho nutnost znalosti velikosti již při kompilaci a nemožnost realokace, což je v tomto případě nežádoucí.

Pro naše potřeby implementace byly tedy vybrány datové struktury dvojrozměrné dynamicky alokované pole a ze standardních knihoven vector. Klasické pole bude využito všude tam, kde jsme schopni zjistit nebo odvodit velikost tohoto pole, které je třeba alokovat. Vector bude poté využit pouze u komprimace, kdy nejsme schopni určit ani odhadnout jakou délku budou mít jednotlivé komprimované sloupce. Je tedy zřejmé,

že u generování a dekomprimace využijeme jednoduchost a rychlost klasického pole, zatímco u komprimace s výhodou využijeme automatickou změnu velikosti vectoru. [24]

5.3.1 Vector

Vector, je STL kontejner pro sekvenční uchovávání prvků s automatickou změnou své velikosti. Vnitřní strukturou je vector dynamicky alokované pole, které pokud se zcela zaplní, je nutné realokovat pro zvýšení jeho kapacity a možnost přidávat nové prvky. Tato operace je relativně náročná na čas zpracování, a proto vector vždy alokuje místo navíc pro případný další růst. Různé implementace mohou k tomuto růstu při realokování přistupovat odlišně s ohledem na vyvážení použitého místa a počtem nutných realokací. Díky tomu, že realokace nemusí probíhat pokaždé, když přidáme nový prvek zabírá vector větší místo než je skutečný počet v něm uložených prvků, a je tedy možné pomocí metody push_back( ) s konstantní časovou složitostí. Tuto velikost můžeme zjistit pomocí metody capacity( ) a zjistit tak kolik prostoru zbývá pro přidávání prvků a také kdy proběhne realokace.

Standardní předpis vectoru je template <class T, class Alloc = allocator<T>>class vector; kdy jako první argument vystupuje datový typ, který bude ukládán a druhý volitelný parametr umožňuje zadat vlastní alokátor, v opačném případě bude použit standardní, používaný všemi standardními kontejnery.

Při vytváření vectoru je k dispozici několik druhů konstruktorů, které lze využít.

```
Vector<DataType> v;
Vector<DataType> v1(v2);
Vector<DataType> v(n);
Vector<DataType> v(n, val);
Vector<DataType> v(start, stop);
```

Výpis 2: Vector - konstruktory

Prvně uvedený konstruktor pouze vytváří prázdný vektor bez jakýchkoliv prvků. Druhý vytvoří vector v1 a naplní jej kopií vectoru v2 za předpokladu, že vector v2 obsahuje prvky stejného typu. Třetí konstruktor vytváří vektor obsahující n prvků vytvořených podle jejich defaultního konstruktoru. Čtvrtý pracuje obdobně a vytvoří n prvků podle druhého parametru. U primitivních typů prvky přímo zadané hodnoty, u objektů vector naplní podle definice argumentu val. Poslední uvedený naplní vector hodnotami v rozsahu start včetně, po stop.

Dále obsahuje některé užitečné metody pro práci s vectorem, jako již zmíněný push_back pro vkládání prvků na konec vectoru. Opakem pak je metoda pop_back, které provádí smazání posledního prvku. Další z metod zmíníme size a empty, přičemž metoda size umožňuje vrácení počtu uložených prvků. U metody empty probíhá test velikosti vectoru, a pokud je velikost nulová vrací true, v opačném případě vrací false. Poslední metodu, kterou zmíníme je metoda clear, která odstraní všechny prvky obsažené ve vectoru, což

[24] STL Containers. [online]. [cit. 2014-03-10]. Dostupné z: http://msdn.microsoft.com/en-us/library/1fe2x6kt.aspx

je vhodné zejména pokud nám slouží pouze, jako dočasné úložiště, které má být opět plněno novými prvky. [25]

5.3.1.1 Vector <bool> Vector <bool>je specializovaným případem vectoru. Jeho hlavním cílem, obdobně jako u bitsetu, je ukládat prvky typu bool, jako jednotlivé bity přímo do paměti. Jeho chování je obdobné jako u nespecializovaného vectoru s určitými rozdíly vztahující se k této specializaci. Hlavní je si uvědomit, že ve skutečnosti nevytváří pole datového typu bool, ale pouze nastavuje bity v paměti a pro přístup k nim používá speciální referenční typ, jako tomu bylo u bitsetu. Navíc přidává metodu flip, která způsobí, že všechny hodnoty true ve vectoru se změní na false a naopak.

Nicméně má i své nevýhody, kdy generický vector<T>bude fungovat pro všechny datové typy kromě bool, a to kvůli nemožnosti referencovat či dereferencovat jednotlivé prvky takovéhoto vectoru, protože ve skutečnosti se nejedná o datový typ bool, ale bity se kterými nedokáže adresovat. Také proto se objevovali výzvy k odstranění této specializace ze standardu nebo alespoň jejího přejmenování. A bylo doporučeno jej v těchto případech nepoužívat. [26]

5.3.2 Generování

Prvním krokem důležitým pro to abychom mohli komprimovat, jsou vstupní data. Pro naše účely budeme tato data generovat a simulovat tak data reálná. Budeme tedy vytvářet bitmapový index námi zvolené velikosti, kdy sloupce budou odpovídat hodnotám, kterých může nabývat indexovaný atribut, tudíž počet vygenerovaných sloupců bude odpovídat počtu unikátních hodnot, kterých atribut nabývá. Řádky v tomto případe, budou reprezentovat počet uložených záznamů v tabulce, kde se tento indexovaný atribut nachází, a tedy počet řádků indexu bude odpovídat počtu záznamů v tabulce.

Jak bylo uvedeno vybraný datový typ, který nám bude představovat jednotlivé bity indexu byl zvolen char a jako struktura udržující všechny chary pohromadě dynamicky alokované pole. Jelikož je bitmapový index matice bitů je třeba jej udržovat v dvojrozměrné formě jako dvojrozměrné pole. Je to tedy ukazatel na pole ukazatelů, které dále ukazují na pole charů. Názorněji to lze vidět na výpisu 3. Takhle dosáhneme alokování potřebného místa pro náš bitmapový index.

Další z faktorů nutných pro tuto funkci je předání velikosti generovaného pole, aby bylo možné jej vytvořit a naplnit daty. Tuto informaci funkce získá z dotazu vypsanému uživateli na obrazovku. Takto získané hodnoty jsou samozřejmě počty bitů indexu, a tedy u počtu řádku je nutné toto číslo převést odpovídajícím způsobem na počet bajtů z důvodu, že jeden char zaujímá osm bitů. S těmito informacemi již můžeme vytvořit samotnou strukturu. Využijeme také toho, že většina bitů takovéhoto indexu bude nulová. Po alokování potřebného místa tedy stačí projít jednotlivě celé toto pole a celé jej nastavit na hodnotu 0x00 neboli všechny jeho bity na nuly. Tímto ušetříme potřebu nastavovat

[25] Std::vector. [online]. [cit. 2014-03-10]. Dostupné z: http://www.cplusplus.com/reference/vector/vector/

[26] Std::vector<bool>. [online]. [cit. 2014-03-10]. Dostupné z: http://www.cplusplus.com/reference/vector/vector-bool/

všechny bity, ale bude nám stačit pouze nastavovat bity, které mají být nastaveny na jedničku.

```
char** index = new char*[c];
for(int i = 0; i < c; ++i)
{
   index[i] = new char[row];
}

for ( int i = 0; i < c; i++ )
{
   for (int j = 0; j < row; j++)
   {
      index[i][j]=0x00;
   }
}
```

Výpis 3: Ukázka alokace dvojrozměrného pole a jeho naplnění počátečními hodnotami

Po vytvoření takového indexu obsahujícím samé nuly je třeba pro každý řádek nastavit jeden bit na jedničku. Abychom dosáhli naplnění bez toho, abychom jednotlivé nastavované bity určovali ručně, použijeme funkci rand () generující pseudonáhodná čísla v rozsahu [0,c) pro určení, do kterého sloupce se má zapsat jednička v aktuálním řádku. Pro projití všech řádků indexu je třeba dvou cyklů. Vnitřní určující aktuálně nastavovaný řádek, tedy bit ve znaku, kde aktuálně procházený znak určuje cyklus vnější. V každém průchodu je pak vygenerováno pseudonáhodné číslo určující sloupec a pomocí bitového posunu a bitové operace OR se v daném sloupci a znaku nastaví jednička na pozici udanou vnitřním cyklem. Zobrazeno ve výpisu 4

```
for ( int i = 0; i < row; i++ )
{
      for (int j = 7; j >= 0; --j)
      {
         random = rand()%c;
         index[random][i]|= 1 << j;
      }
   }
```

Výpis 4: Nastavení jedniček bitmapového indexu.

5.3.3 Komprimace

Když již máme k dispozici data ve vhodné podobě, můžeme přikročit již k samotné komprimaci. Samotný postup komprimace byl již popsán v kapitole 3.3, a proto představíme pouze konkrétní implementaci tohoto postupu. Prvním krokem, který jsme již splnili, jsou vhodná data. Ty jsme získali tím, že jsme alokovali pole velikosti načítaného indexu a následně do něj načetli tento index ze souboru ve kterém je uložen. Následně je nutné připravit si strukturu po zpětné ukládání již komprimovaného vectoru a to jak strukturu pro celý komprimovaný index, tak i vector pro jednotlivé komprimované sloupce, které se následně budou přidávat do struktury indexu. Další důležité proměnné jsou unsigned

char c, jakožto základní člen pro ukládání jednotlivých bitů, a int position pro uchovávání aktuální pozice v nastavovaném znaku. Ještě je nutno podotknout, že znak c je nastaven na hodnotu 0xFF hexadecimálně, binárně tedy 11111111 a to z toho důvodů, že ve výsledném komprimovaném indexu budou značně převládat jedničky a vyhneme se tak zbytečně častému nastavování jednotlivých bitů a místo toho bude potřeba pouze nulovat dané bity. Druhým důvodem je pak fakt, že dekomprimace probíhá na principu sledování sekvence jedniček následovaných nulou, a tedy zbytkové bity (kterých může být až 7) nepatřící již k indexu, ale jsou způsobeny velikostí sloupců indexu, která není přesně násobkem osmi, nám neovlivní následnou dekomprimaci a nezanesou nám chybu. Poslední potřebné proměnné jsou int count pro uchovávání počtu nul v aktuální sekvenci.

K samotné komprimaci je poté třeba procházet jednotlivé bity indexu a to tak, že jej budeme procházet po jednotlivých sloupcích, daný sloupec pak po jednotlivých znacích a nakonec každý znak po jednotlivých bitech. Při průchodu znaky po jejich bitech se do proměnné count přičítá vždy jednička pokud daný bit nabývá hodnoty nula.Ve výpisu 5 je vidět, že pro kontrolu bitů indexu je použit bitový posun doleva o k, masky hodnoty 1 (00000001) a operace AND. Proměnná i značí sloupec, j znak a k bit ve znaku. V případě, že se narazí na jedničku dosáhli jsme konce jedné sekvence nul a je třeba ji tedy zpracovat.

```
if (!( index[ i ][ j ]&(1 << k)))
   {
      count++;
   }
```

Výpis 5: Kontrola jednotlivých bitů ve znaku.

Zpracování probíhá tak, že prvně je třeba zjistit počet bitů vyjadřující hodnotu proměnné count, určující nám počet nul před dosažením jedničky. Toto zjištění nám zajistí funkce BitCount, která nám vrátí počet bitů. S těmito informacemi můžeme začít tvořit výsledný komprimovaný vector ze znaků c. Víme, že znak c obsahuje pouze osm bitů a musíme kontrolovat při každém posunu, zdali jsme neprošli již všechny bity. V takovém případě je potřeba znak uložit do vectoru sloužící jako dočasný sloupec indexu pomocí metody push_back(c) a opět znak c nastavit na hodnotu 0xFF, společně s nastavením ukazatele pozice ve znaku opět na nejvyšší bit tedy na 7. Když již máme zajištěnu tuto kontrolu, můžeme začít správně nastavovat bity ve znaku, tak jak je třeba. První operací je třeba nastavit $n-1$ bitů na jedničku, kde n je počet potřebných bitů k vyjádření čísla count a tento počet již víme z funkce BitCount. Díky tomu, že nastavovaný znak má na začátku všechny bity nastaveny na jedničku, stačí nám dekrementovat pozici ve znaku představovanou proměnnou position a nemusíme tak bity nastavovat a v případě projití všech bitů (position<0) se provedou operace popsané výše a zobrazeny ve výpisu 6. Poté co je zachován potřebný počet jedniček je třeba negovat jeden následující bit, k tomu je využita operace XOR. Posledním krokem je zapsání samotné binární reprezentace čísla count. K tomu slouží samostatný cyklus, který prochází po jednotlivých bitech proměnnou count, od nejvyššího nastaveného bitu, tedy BitCount-1. Takto čteme bity od nejvyššího a v případě jedničky se opět dekrementuje position, v případě nuly se pomocí XOR operace bit vynuluje. Samozřejmostí je stále kontrola pozice ve znaku. Takto je zpracována první sekvence nul a je třeba vynulovat jejich počítadlo a nastavit count zpět na nulu.

V případě, že byl již zpracován celý vstupní sloupec, mohou nastat dva stavy. Jeden z nich nastává v případě, že daný sloupec neobsahoval žádnou jedničku a tedy výsledný komprimovaný sloupec neobsahuje žádný znak a působil by tedy problém při dekomprimaci, kdy by jeden sloupec ve výsledku chyběl. Druhý případ nastává, pokud délka sloupce není násobkem osmi a několik posledních bitů posledního znaku zůstane nastaveno na jedničky. Toto chování je žádoucí, problém je pouze v neuležení tohoto znaku naší funkcí kontrolující pozici ve znaku. V obou případech je tak třeba zapsat znak do vectoru sloužícího jako komprimovaný sloupec. V prvním případě tak bude zapsán znak se samými jedničkami, v případě druhém znak zčásti nastaven. Následuje nastavení proměnných do počátečního stavu pro použití při zpracování dalšího sloupce, uložení komprimovaného sloupce do struktury tomu určené a jeho následné vyprázdnění.

Po provedení pro všechny sloupce vstupního bitmapového indexu jsme získali jeho komprimovanou podobu, která je připravena k zápisu do souboru.

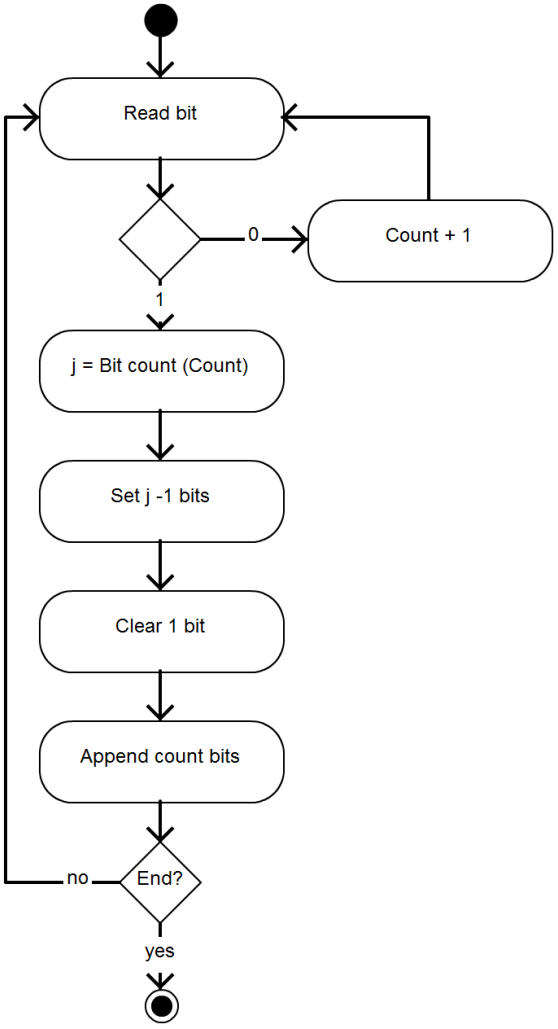
```
if (position<0)
    {
      ComSloupec.push_back(c);
      position=7;
      c=0xff;
    }
```

Výpis 6: Kontrola pozice ve znaku a operace při zaplnění celého znaku.

5.3.4 Dekomprimace

Nedílnou součástí programu na komprimaci je také možnost dané data správně dekomprimovat. Proto si přestavíme funkci, která námi komprimovaný bitmapový index opět dekomprimuje. Tak jako v případě komprimace budeme potřebovat strukturu na ukládání dekomprimovaného indexu, tedy dynamicky alokované dvojrozměrné pole, kdy bude velikostí odpovídat originálnímu indexu a všechny jeho znaky budou nastaveny na 0x00. K tomuto poli také potřebujeme další dynamicky alokované pole pro uložení načteného komprimovaného indexu a jedno pole pro uložení délek jednotlivých komprimovaných sloupců. Dále nutné proměnné typu int a to count pro načítání sekvencí jedniček, na začátku nastavena na 0, bit pro určení pozice ve znaku komprimovaného indexu a position pro určení pozice ve znaku sloužícího jako v případě komprimace pro složení výsledného indexu. Poslední proměnnou typu int je poté number udržující výsledný počet nul originálního indexu. Pro postupné sestavení potřebujeme unsigned char c, ve kterém budeme postupně nastavovat potřebné bity, v tomto případě zpočátku nastaven na 0 z důvodu převládajících nul a snížení počtu nutných nastavování bitů.

Předpokládejme, že komprimovaná data máme již k dispozici v paměti a to buď z předchozí komprimace, nebo jsou již načtena za souboru. Při dekomprimaci budeme postupovat obdobně jako při komprimaci s tím rozdílem, že nyní pro nás budou stěžejní sekvence jedniček následovaných nulou. Je tedy potřeba tří cyklů na řádné projití všech znaků komprimovaného indexu. Hlavním cyklem bude cyklus na procházení bitů jednotlivých znaků. Je tedy důležité udržovat si aktuální pozici ve znaku a v případě potřeby přejít plynule na znak další.



Obrázek 6: Komprimace.

Prvním krokem tedy je po jednotlivých znacích kontrolovat hodnoty jejich bitů na výskyt jedničky. Pokud se tak stane a jednička je nalezena inkrementuje se počítadlo a kontrolují se bity do té doby, než se naleze první nula. Při nalezení takovéto sekvence je zjištěno kolik následujících bitů má být přečteno a ty nám vyjádří počet nul následovaných jedničkou, jenž je třeba zapsat na výstup pro rekonstruování originálního indexu. Abychom toho dosáhli potřebujeme další cyklus začínající od nuly a končící číslem count, takto zajistíme, že načteme $count + 1$ následujících bitů. Pro zajištění pokračování na správné pozici po načtení těchto bitů budeme ovlivňovat proměnné cyklů určující nám polohu, jak v aktuálním znaku, tak případně posouvat ukazatel znaků samotných. Nejdříve je nezbytné posunout ukazatel na bit, abychom nečetli znova ten, který sloužil jako zarážka po sekvenci jedniček. Pomocí operace AND a bitového posunu tedy čteme následující bit a určujeme jeho hodnotu. V případě, že nabývá hodnoty jedna je proměnná number pomocí operátoru $<<$ posunuta o jedno doleva a následně nastaven první bit na jedničku. V případě nuly je proměnná number pouze posunuta o jedno vlevo. Po načtení všech těchto bitů máme v proměnné number počet nul následovaných jedničkou.

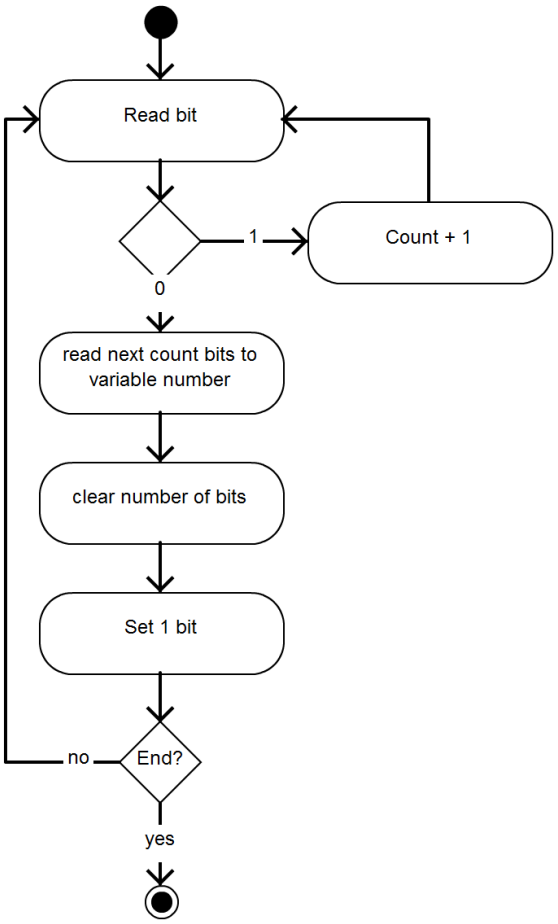
Nyní začneme vytvářet jednotlivé znaky tvořící dekomprimovaný index. Jelikož na počátku je znak c vždy nastaven na 0, není třeba zapisovat nuly, ale jen dekrementovat proměnnou position určující pozici ve znaku. Při snížení její hodnoty pod 0 je potřeba daný znak zapsat na správnou pozici v poli sloužící pro uložení dekomprimovaného indexu. Po zapsání potřebného počtu nul nastavíme následující bit na jedničku a zkontrolujeme pozici ve znaku, výpis 7. Následuje vynulování počítadla jedniček count a vynulování proměnné number. V případě, že projdeme všechny bity nastavovaného znaku uložíme jej a inkrementujeme proměnnou určující aktuální řádek v dekomprimovaném indexu.

```
c |= 1 << position;
position--;
if (position<0)
{
    output[OutputColumns][OutputRows]=c;
    OutputRows++;
    position=7;
    c=0;
}
```

Výpis 7: Nastavení bitu na jedničku a kontrola pozice ve znaku.

Tímto proběhlo rozklíčování jedné části komprimovaného bitmapového indexu a je třeba jej opakovat pro celý sloupec a všechny jeho znaky a tedy všechny bity. Pokud se tak již stalo dochází ke kontrole, zdali není takto dekomprimovaný sloupec prázdný k čemuž by došlo v případě, že by neobsahoval žádnou jedničku a zároveň kontrola proměnné position. Pokud position nebude rovno 7 je zřejmé, že nebyl zaplněn celý znak, a nebyl tedy zapsán do výsledného sloupce, a je tak tedy třeba učinit. V případě, že daný sloupec neobsahuje žádnou jedničku nemusíme se o něj starat z důvodu, že na začátku byly všechny prvky pole sloužícího pro dekomprimovaný index nastaveny na 0. V posledním kroku zbývá nastavit potřebné proměnné do výchozího stavu.

Po vykonání všech potřebných operací a obnovení originálního indexu je třeba z paměti uvolnit alokované místo po komprimovaný index a jeho délky. To zajistíme pomocí



Obrázek 7: Dekomprimace.

delete [] viz. výpis 8. Nejprve je třeba odstranit v cyklu jednotlivé sloupce komprimovaného indexu a následně celý index. Poté zbývá odstranit pole sloužící pro uložení délek sloupců. V případě, že by jsme dále nepracovali s dekomprimovaným indexem bylo by třeba stejným způsobem odstranit také toto pole.

```
for(int i = 0; i < CommpressCol; ++i)
{
   delete [] index[i];
}
delete [] index;
delete [] lenghts;
```

Výpis 8: Uvolnění alokovaného pole.

Z důvodu přednastavení pole pro dekomprimovaný index na potřebou velikost a jeho nastavení na nuly dokážeme získat celý původní bitmapový index. Z principu dekomprimace totiž vyplývá nemožnost jeho plného obnoveni. To znamená, že veškeré nuly za poslední jedničkou v každém sloupci by se neobnovili. Tímto jsme tedy eliminovali tento nežádoucí jev.

5.3.5 Dekomprimace zadaného sloupce

Tato funkce probíhá obdobně jako výše popsaná dekomprimace s tím rozdílem, že se provádí pouze nad jedním zadaným sloupcem indexu. To přináší výhodu v tom, že při operacích s indexem nepotřebujeme dekomprimovat index jako celek, ale pouze jeho potřebné části. Opět je tedy do paměti načten komprimovaný index a délky jeho jednotlivých sloupců. Je tedy alokováno pole index potřebné velikosti a postupně do něj načítány jednotlivé sloupce indexu. Do druhého alokovaného pole jsou zároveň načítány délky jednotlivých sloupců. Toto načtení je ukázáno ve výpisu 9.

Samotná dekomprimace poté probíhá stejně jako v předchozím případě, ale pouze pro zvolený sloupec. Není tedy třeba pro něj alokovat dvojrozměrné pole. Alokujeme tedy pouze pole potřebné velikosti a opět všechny jeho prvky nastavíme na 0x00. Následně do tohoto pole ukládáme po částech dekomprimovaný sloupec. Po provedení celé dekomprimace pak pole output obsahuje dekomprimovaný sloupec pro další použití. Nesmíme poté zapomenout opět uvolnit alokovanou paměť, kterou jsme zabrali poli pro komprimovaný index a jeho délky opět dle výpisu 8

```
char** index;
int* lenghts;
length.open("delky_komp.txt");
fopen_s(&file,"komprimat.bin","rb");
length>>CommpressCol;
index = new char*[CommpressCol];
lenghts = new int[CommpressCol];
for(int i = 0; i < CommpressCol; ++i)
{
   length>>CommpressRow;
   index[i] = new char[CommpressRow];
   lenghts[i]=CommpressRow;
```

```
    fread(index[i], sizeof(unsigned char), sizeof(unsigned char)*CommpressRow,file);
}
fclose( file );
length.close();
```

Výpis 9: Načtení komprimovaného indexu do paměti.

5.3.6 Kontrola stavu na zadané pozici

Další z nabízených funkcí je kontrola, zda se na zadané pozici v indexu nachází jednička nebo nula. Po zadání pozice v indexu kterou chceme zkontrolovat se provádí dekomprimace v daném sloupci a kontrolují se pozice na kterých se vyskytují jedničky. Probíhá tedy normální dekomprimace sloupce, ale bez toho aby se tento sloupce ukládal v paměti.

Opět prvně musíme načíst komprimovaný index do paměti i s délkami jeho sloupců. A je provedena kontrola zda je zadaná pozice v indexu platná. Poté již probíhá dekomprimace. Místo pole pro výstup dekomprimovaného indexu nebo jeho části, zde potřebujeme pouze proměnou udržující pozici nalezených jedniček typu int one_position. Druhou potřebnou proměnnou je pak bool found která značí zda byla na zadané pozici jednička. Pokud ano nastaví se na true a není třeba dále dekomprimovat.

Pokud při dekomprimace zjistíme přítomnost jedničky uložíme tuto pozici do one_position. Každé další nalezení jedničky znamená počet nul mezi těmito jedničkami a je tento počet třeba přičíst k one_position aby jsme zjistili pozici další jedničky. Po každém takovémto nalezení jedničky je prováděna kontrola zda pozice jedničky odpovídá zadané pozici, kterou chceme kontrolovat. Pokud se tak stane nastavím found na true, vypíšeme upozornění, že na této pozici se jednička skutečně nachází a ukončíme dekomprimaci. Dekomprimace končí také v tom případě, že pozice nalezené jedničky je již větší než námi kontrolované. je tedy zbytečné pokračovat v dekomprimaci. V případě, že jednička na dané pozici nalezena nebyla a tudíž found je stále nastaveno na false je vypsáno upozornění, že na dané pozici se jednička nenachází. Nyní zbývá již pouze odstranit z paměti načtený a alokovaný komprimovaný index.

5.3.7 Další funkce

Ke správné funkčnosti je třeba také funkce potřebná ke zjištění počtu bitů čísla což je nutné při komprimaci. Tato funkce pracuje za pomoci cyklu do-while kdy jsou postupně posouvány bity vstupního čísla o 1 vpravo. Díky těmto posunům se nám vždy ztratí nejnižší bit čísla a můžeme tedy posouvat do té doby než bude číslo nulové. Při těchto posunech počítáme jejich počet, který nám poté značí kolik je pro dané číslo třeba bitů. Cyklus končí v případě, že již posouvané číslo je nula a funkce poté vrací počet jeho bitů. Tato funkce je zobrazena na výpisu 10

```
int count=0;
do
{
    count++;
    length >>= 1;
```

```
}
while ( length );
return count;
```

Výpis 10: Spočítání počtu bitů čísla.

Další z nutných funkcí jsou funkce pro zápis a čtení dat. Při zápisu je nutné si uvědomit, že zapisujeme sekvenci znaků, a je třeba tedy zapisovat výhradně v binárním režimu. Při použití textového režimu by se mohlo snadno stát, že nějaký ze zapisovaných znaků bude odpovídat jednomu z kontrolních znaků ASCII a ovlivnili by pak výsledný zápis, a to by vedlo k nemožnosti správného načítání dat. Druhou nezbytnou součástí je zapsání druhého souboru obsahujícího metadata ve formě délky jednotlivých zapisovaných sloupců pro správné vytvoření struktury při čtení.

```
length.open("delky_komp.txt");
ofstream compout("komprimat.bin", ios::out | ios::binary);
for (unsigned int i=0; i<index.size(); i++)
{
    compout.write((const char*)&index[i][0], index[i].size());
    length<<index[i].size()<<"\n";
}
```

Výpis 11: Otevření souboru a zapsání komprimovaného indexu.

Při zápisu je nejprve nutné vytvořit stream umožňující zápis do souboru a zvolit jeho režim jako výstupní a binární. Tak jak je uvedeno ve výpisu 11, kdy je vytvořen stream compout, který bude binárně zapisovat do souboru komprimat.bin. Následuje již samotné zapsání jednoho celého sloupce, a je tedy třeba tento zápis provést pro všechny sloupce indexu. Metoda write přímá jako argumenty ukazatel na pole znaků a velikost tohoto pole, neboli počet prvků tohoto pole, které chceme zapsat, a provádí kopii tohoto bloku dat do souboru. Předáme ji proto ukazatel na první prvek vectoru reprezentující nám sloupec a jeho velikost značící kolik znaků obsahuje. V případě, že zapisujeme teprve vytvořený index je pouze rozdíl v tom, že musíme znát počet sloupců a řádku indexu. Po zapsání vlastních dat sloupce je následně zapsána do souboru metadat jeho velikost. Po zapsání všech sloupců je soubor a stream uzavřen metodou close().

Při načítání ze souboru potřebuje před samotným načtením vyhradit dostatek místa v paměti pro tyto data, a tedy alokovat potřebné místo. Velikost tohoto potřebného místa nicméně musíme nejprve zjistit ze souboru samotného. K tomu využijeme soubor obsahující velikost indexu, buď komprimovaného nebo originálního. Prvně tedy načteme počet sloupců, které index obsahuje a alokujeme pole jak pro index tak pro délky sloupců. Poté je nutné v cyklu načítat jednotlivé sloupce. Načteme tedy velikost prvního sloupce. pomocí této velikosti alokujeme v poli místo pro tento sloupec a samotnou velikost uložíme do pole lenghts. Funkcí fread nakonec provedeme samotné čtení dat do námi alokované paměti. Předáme ji tedy ukazatel na tuto paměť společně s velikostí jednotlivých prvků načítaných ze souboru v bajtech, v našem případě velikost unsigned char, počet takovýchto prvků a jako poslední ukazatel na soubor, z něhož se má číst. zobrazeno ve výpisu 12

```
length>>CompressCol;
index = new char*[CompressCol];
lenghts = new int[CompressCol];
for(int i = 0; i < CompressCol; ++i)
{
    length>>CompressRow;
    index[i] = new char[CompressRow];
    lenghts[i]=CompressRow;
    fread(index[i],sizeof(unsigned char), sizeof(unsigned char)*CompressRow,file);
}
```

Výpis 12: Zjištění velikosti dat v souboru, alokování potřebné paměti a načtení.

Nyní, když jsou data správně načtena, je nutné zavřít oba streamy jak pro načtení dat, tak i pro načítání délek sloupců.

6 Testování

Při testování vytvořené aplikace je nutné se v první řadě zabývat správností prováděné komprimace, tak i dekomprimace. Dalšími sledovanými aspekty budou velikosti vstupních a výstupních dat a jejich poměr, tedy kompresní poměr, kterého jsme touto metodou schopni dosáhnout a časy jednotlivých dílčích úloh prováděných v programu. Veškerá testování budou probíhat na referenčním notebooku značky ASUS s parametry uvedenými níže. Lze tedy předpokládat, že na různě výkonných počítačích budou výsledné časy odlišné, zatímco velikosti vstupních a výstupních dat nikoliv. Pro testování budou vytvořeny různé vzorky bitmapového indexu o různých velikostech pro reprezentaci různé hustoty jedniček v něm obsažených.

6.1 Správnost komprimace a dekomprimace

Pro určení správnosti komprimace daného bitmapového indexu nemáme žádný programový prostředek, kterým bychom byli schopni určit, zdali došlo ke komprimaci správně. Jediným východiskem v této situaci je vizuální kontrola v debugovacím režimu vývojového prostředí, která je při velkém objemu dat téměř nemožná. Je tedy nutné správné fungování implementovaného algoritmu ověřit na relativně malém vzorku dat, který je člověk ještě schopen zkontrolovat. Při této kontrole je třeba jednotlivé znaky tvořící nám komprimovaný bitmapový index převést na jednotlivé bity je tvořící, a zkontrolovat s námi určenou sekvencí jedniček a nul, kterou jsme získali ručním převodem z nekomprimovaného indexu.

Při kontrole dekomprimačního algoritmu toto úskalí již není třeba podstupovat, i když vizuální kontrola na malém vzorku dat je stále možná. V tomto případě můžeme jednoduše využít znalosti původního nekomprimovaného bitmapového indexu. Samotná kontrola pak může probíhat dvěma způsoby a to buď porovnáváním celých znaků, z nichž se skládají dekomprimovaný a originální index nebo kontrolou počtu dekomprimovaných jedniček oproti originálnímu indexu a jejich pozicí v indexu.

1. Porovnávání celých znaků

   Při tomto druhu ověření je třeba mít v paměti zaveden originální i dekomprimovaný index, aby vůbec mohla proběhnout jejich kontrola. Samotná kontrola pak spočívá v porovnávání znaků dekomprimovaného indexu se znakem na odpovídající pozici originálního indexu. V případě neshody i jediného z těchto znaků to znamená, že se dekomprimace provedla nesprávně a bylo by třeba upravit dekomprimační algoritmus.

2. Kontrola počtu a pozice jedniček

   Pro provedení této kontroly správnosti je hlavním cílem projít všechny jednotlivé bity obou indexů a zaznamenat počty jedniček vyskytujících se v nich. Pokud tyto dva počty jsou si rovny, můžeme přikročit k druhému kroku a to ke kontrole pozic těchto jedniček v obou indexech. Pokud při průchodu dekomprimovaným indexem narazíme na jedničku, je nutné ověřit, zda se jednička nachází na stejné pozici

Model	ASUS K53SV
Operační systém	Windows 7
Procesor	Intel Core i5-2430M 2,4GHz
Operační paměť	4GB DDR3 1333 MHz
Pevný disk	500GB 5400 ot/min

Tabulka 6: Parametry referenčního notebooku

také v originálním indexu. Jelikož jsme již zjistili, že počet jedniček v obou indexech je stejný, je třeba aby všechny jedničky nalezené v dekomprimovaném indexu byly na stejných pozicích také v originálním indexu a poté můžeme prohlásit, že dekomprimace proběhla korektně.

V obou případech je nezbytné, aby meze cyklu procházejícího jednotlivé znaky nebo bity byly nastaveny podle dekomprimovaného indexu z důvodu popsaného výše v sekci 5.3.4 zabývající se dekomprimací.

Po provedení těchto testů můžeme prohlásit, že komprimační i dekomprimační algoritmy byly implementovány správně. Tedy při komprimaci ani dekomprimaci nevznikají chyby z pohledu ztráty nebo nežádoucích změn pozic jedniček bitmapového indexu.

6.2 Vstupy a výstupy

Jako vstupní data pro komprimaci byla vygenerována data o různých velikostech a hustotě rozložení jedniček. Pro srovnání jsou vytvořena data několika velikosti. Byly vytvořeny indexy s různou délkou sloupců (to znamená různým počtem řádků) a to 1000, 100 000 a 10 000 000. Ke všem těmto délkám bylo také nutno zahrnout různé kardinality dat jak můžeme vidět v tabulce 8. V prvním řádku vidíme zvolené stupně kardinality, zatímco v prvním sloupci poté zvolené počty řádků. Samotná data v tabulce poté značí velikost komprimovaného indexu. Růst velikosti nekomprimovaných dat je ve všech případech délek sloupců přibližně lineární jak můžeme vidět na obrázcích 8, 9 a 10. Jako vstup pro dekomprimaci pak slouží tato komprimovaná data. Z těchto uvedených hodnot můžeme určit pro jednotlivá data kompresní poměr, kterého se nám podařilo touto metodou docílit.

$$\text{kompresní poměr} = \frac{\text{velikost komprimovaných dat}}{\text{velikost nekomprimovaných dat}}$$

Z výsledků vidíme, že nejhoršího kompresního poměru, a tedy i nejmenší úspory míst jsme dosáhli u dat, kdy indexovaný atribut má relativně malou kardinalitu a index tedy obsahuje malý počet sloupců s častým výskytem jedniček. Z toho plyne, že v takovém sloupci se bude vyskytovat velké množství krátkých sekvencí k zakódování a komprimace nebude tak efektivní. Například u kardinality atributu 4 je úspora místa naprosto zanedbatelná a bylo by tedy efektivnější používat v tomto případě nekomprimovaný index. Se stále se zvyšující kardinalitou se již kompresní poměr značně zlepšuje a dosahujeme významnější úspory místa. Z grafu na obrázku 11 pak lze snadno vyčíst, že při kompresi nezáleží na délkách komprimovaných sloupců, pokud se jedná o větší než

	4	8	16	32	64
1000	500B	1000B	2000B	4000B	8000B
100000	48,8KB	97,6KB	195,3KB	390,6KB	781,2KB
10000000	4,7MB	9,5MB	19,0MB	38,1MB	76,2MB
	128	256	512	1024	
1000	16000B	32000B	64000B	128000B	
100000	1562,5KB	3125KB	6250KB	12500KB	
10000000	152,5MB	305,1MB	610,3MB	1220,7MB	

Tabulka 7: Velikosti nekomprimovaného indexu pro různou kardinalitu a počty řádků.

	4	8	16	32	64
1000	499B	717B	939B	1191B	1420B
100000	48,5KB	68,9KB	91,2KB	114,8KB	138,9KB
10000000	4,7MB	6,7MB	8,9MB	11,2MB	13,5MB
	128	256	512	1024	
1000	1680B	1943B	2254B	2787B	
100000	163,0KB	187,3KB	211,4KB	236,0KB	
10000000	15,9MB	18,2MB	20,6MB	23,0MB	

Tabulka 8: Velikosti komprimovaných indexů pro různou kardinalitu a počty řádků.

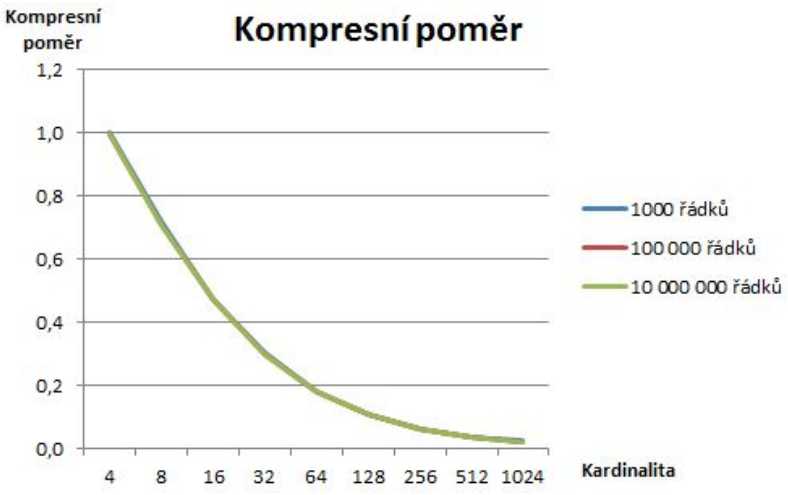
Obrázek 8: Velikost dat při 1000 řádcích.

	4	8	16	32	64	128	256	512	1024
1000	0,998	0,717	0,470	0,298	0,178	0,105	0,061	0,035	0,022
100000	0,993	0,706	0,467	0,294	0,178	0,104	0,060	0,034	0,019
10000000	0,994	0,707	0,467	0,294	0,178	0,104	0,060	0,034	0,019

Tabulka 9: Kompresní poměry pro různou kardinalitu a počty řádků.

Obrázek 9: Velikost dat při 100 000 řádcích.

Obrázek 10: Velikost dat při 10 000 000 řádcích.



Obrázek 11: Kompresní poměry.

malou hodnotu, ale pouze na kardinalitě indexovaného atributu. Můžeme tedy vidět, že s vzrůstající kardinalitou nám význačně klesá potřebné místo k uložení komprimovaného bitmapového indexu.

Takto jsme otestovali různé délky sloupců indexu, tak i různou hustotu rozložení jedniček v indexu. Z výsledků tedy nakonec můžeme usoudit, že délka sloupců nepůsobí na výsledný kompresní poměr. Ten je značně závislý pouze na kardinalitě indexovaného atributu a tedy i na následné hustotě rozmístění jedniček v bitmapovém indexu. Čím nižší je kardinalita bitmapového indexu tím horší dosáhneme výsledné komprese.

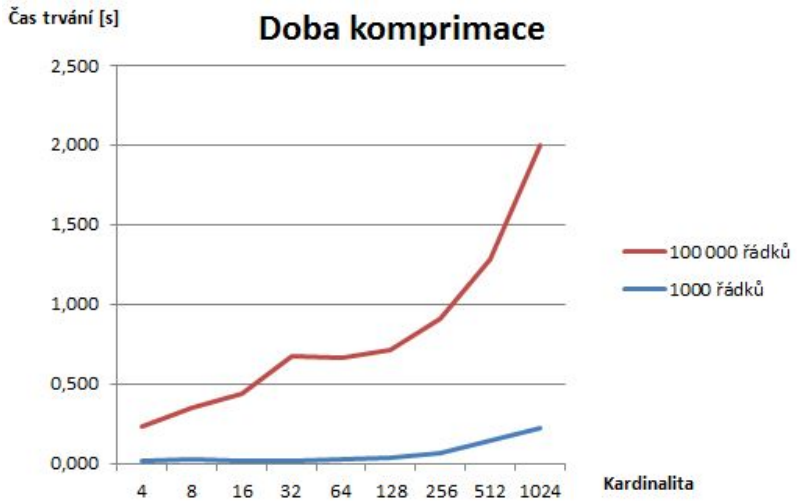
6.3 Čas průběhu

Nedílnou součástí komprimace je také doba, kterou zaberou všechny potřebné operace s tím spojené. Časy jednotlivých operací jsou značně závislé na konkrétním počítači, na kterém jsou prováděny a v našem případě jsou spojeny s referenčním notebookem, na kterém probíhalo testování. V tabulce 10 můžeme vidět doby komprimace pro jednotlivá naše data. Opět tabulka obsahuje časy pro všechna naše data, která jsme měli na testování. Všechny uvedené časy jsou uváděny v sekundách. Jak z časů samotných tak i názorněji na obrázcích 12, 13, kde můžeme vidět grafy průběhu těchto časů, můžeme vyčíst, že časy komprimace jsou závislé jak na kardinalitě tak na délkách jednotlivých sloupců. Jsou tedy vidět, na rozdíl od kompresních poměrů, rozdíly jak při změnách kardinality, tak i délek sloupců. Také můžeme vidět, že i když zdvojnásobíme počet sloupců a tedy kardinalitu, nenastává přímo úměrné zdvojnásobení času komprimace.

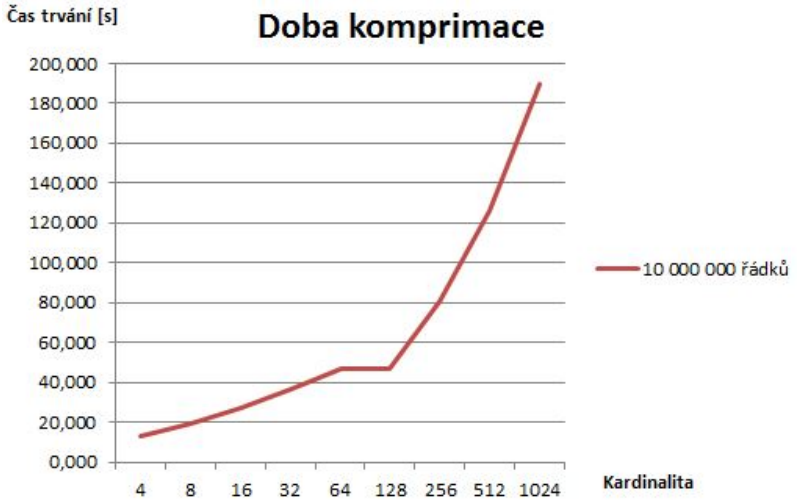
V tabulce 11 poté můžeme vidět časy dekomprimace bitmapových indexů. Tyto indexy odpovídají těm komprimovaným uvedeným výše. První řádek určuje kardinalitu indexu a první sloupec poté délky sloupců originálních dat. Průběhy těchto časů můžeme vidět na obrázcích 14 a 15, všechny časy jsou uvedeny v sekundách. Pokud tyto časy srovnáme s časy komprimace, zjistíme, že jsou značně kratší a tedy dekomprimace probíhá rychleji.

	4	8	16	32	64	128	256	512	1024
1000	0,016	0,022	0,015	0,016	0,028	0,031	0,062	0,140	0,218
100000	0,218	0,327	0,421	0,655	0,639	0,686	0,842	1,138	1,778
10000000	12,714	19,297	27,378	36,332	46,934	47,018	80,231	126,045	189,650

Tabulka 10: Doby trvání komprimace v sekundách pro různou kardinalitu a počty řádků.



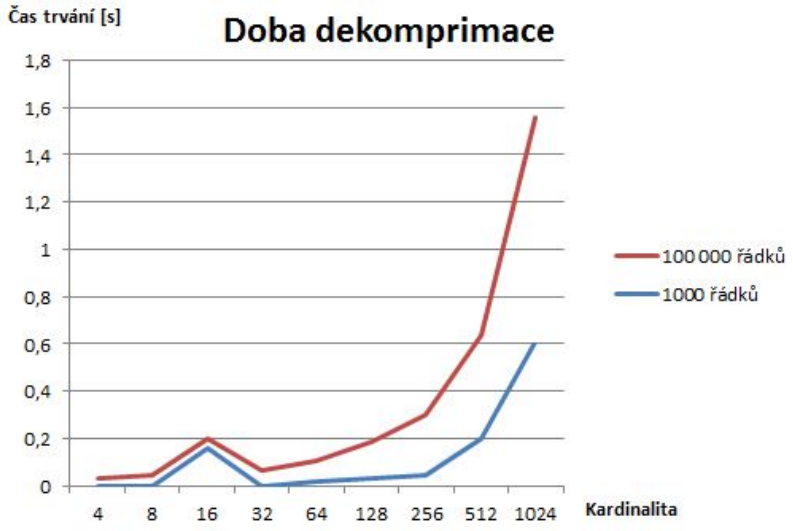
Obrázek 12: Časy komprese pro 1000 a 10000 řádků.



Obrázek 13: Časy komprese pro 10000000.

	4	8	16	32	64	128	256	512	1024
1000	0	0	0,156	0	0,015	0,031	0,048	0,202	0,608
100000	0,031	0,046	0,046	0,062	0,093	0,156	0,249	0,436	0,951
10000000	1,482	1,684	2,48	3,697	6,442	13,026	21,621	34,511	71,104

Tabulka 11: Doby trvání dekomprimace v sekundách pro různou kardinalitu a počty řádků.



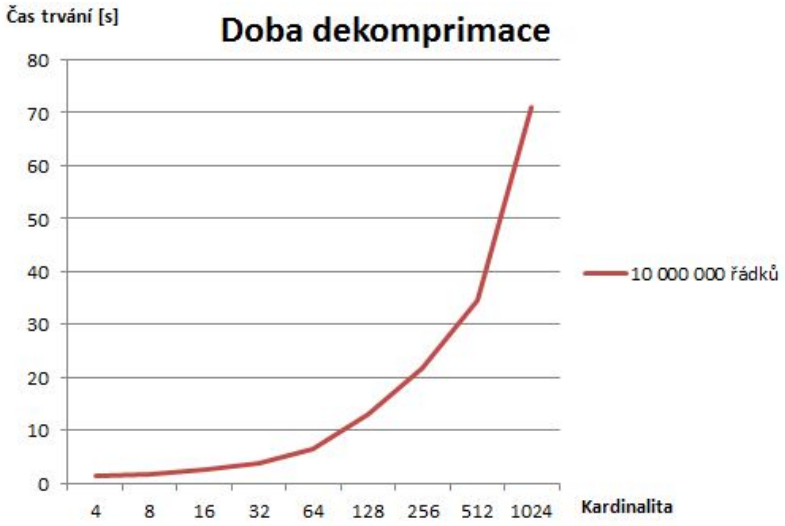
Obrázek 14: Časy dekomprese pro 1000 a 10000 řádků.

To je zapříčiněno tím, že při dekomprimaci již máme vytvořeno pole dostatečné velikosti pro originální index a vkládáme pouze jeho potřebné části. Tato operace je poté rychlejší než vytváření komprimovaného indexu. Z pohledu použitelnosti poté platí, že čím kratší doby dekomprimace, tím více operací dokážeme vykonat nad komprimovaným indexem. je tedy žádoucí aby tyto časy byly co nejkratší.

Nicméně v praxi při používání budeme jen zřídka dekomprimovat celý bitmapový index pro práci s ním. Tímto způsobem by jsme ztratily výhodu jeho velikosti v komprimovaném stavu. Při provádění operací nad tímto indexem budeme nejčastěji pracovat s jednotlivými sloupci nebo zjišťovat zda se na dané pozici indexu nachází jednička nebo nula.

6.4 Dekomprimace sloupců

Důležitou operací určitě bude vyhledávání v komprimovaném bitmapovém indexu. Zde je nárokem provedení co největšího počtu operací, za co nejkratší dobu. Tuto vlastnost jsme testovali na datech o kardinalitě 1024 a tedy index má 1024 sloupců. Tyto data byla reprezentována délkami sloupců 100 000 a 10 000 000 řádků. Probíhalo tak testování za jak dlouho jsme schopni z komprimovaného bitmapového indexu zjistit a tedy dekompri-



Obrázek 15: Časy dekomprese pro 10000000.

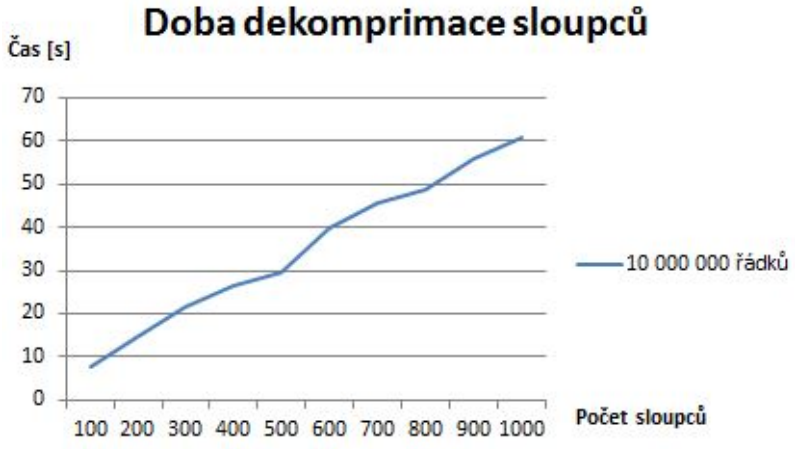
	100	200	300	400	500	600	700	800	900	1000
100000	0,005	0,012	0,021	0,03	0,035	0,047	0,057	0,065	0,066	0,076
10000000	7,85	15,03	21,54	26,61	29,74	39,62	45,68	48,7	55,77	60,97

Tabulka 12: Doby trvání dekomprimování sloupců v sekundách. Pro 100-1000 sloupců a 100 000 a 10 000 000 řádků.

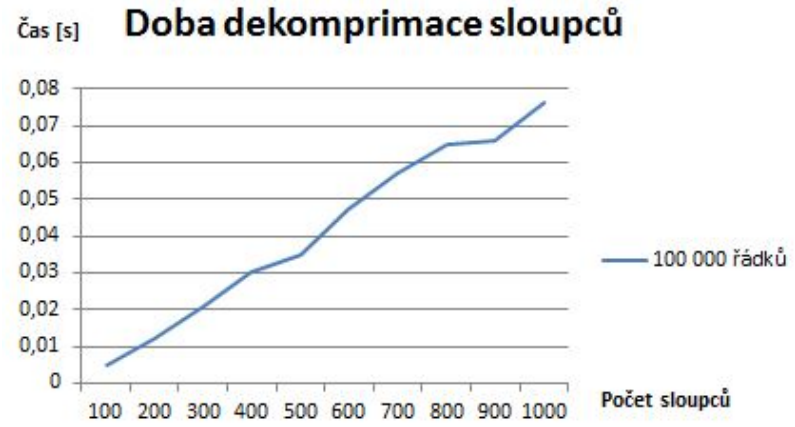
movat jednotlivé sloupce pro další zpracování. Výsledné časy pro daný počet vybíraných sloupců jsou uvedeny v tabulce 12 (časy jsou uvedeny v sekundách). Testování probíhalo tak, že byl zvolen počet sloupců, které chceme najedou dekomprimovat, a tedy provádět s nimi operace, a tyto sloupce byly dekomprimovány. Této dekomprimaci byl měřen čas za jak dlouho se provede dekomprimace všech potřebných sloupců. Nutno podotknout, že komprimovaný bitmapový index byl po celou dobu průběhu uložen v operační paměti a tedy nebylo třeba jej neustále načítat ze souboru.

Při délce sloupců 100 000 řádků vidíme na obrázku 17 časový průběh dekomprimace pro jednotlivé počty dekomprimovaných sloupců. V průměru poté vychází, že za jednu sekundu jsme schopni dekomprimovat přibližně 13 000 sloupců a provést s nimi žádané operace. V dat délky sloupců 10 000 000 je situace již horší. V průměru dokážeme dekomprimovat 15 sloupců za sekundu. Musíme ale také vzít v potaz, že pracujeme s velmi dlouhými sloupci a zpracovávali by jsme 10 miliónů řádků indexu najednou. Časový průběh je opět zobrazen na obrázku 16. Z obou grafů můžeme vyčíst, že časy dekomprimace při vzrůstajícím počtu sloupců stoupají přibližně lineárně. Z toho můžeme usoudit, že nepříliš záleží na rozložení jednotlivých jedniček v konkrétních sloupcích.

Druhým faktorem těchto uvedených dob je výkon počítače na kterých jsou tyto operace prováděny. Na výkonném databázovém serveru budou tyto časy značně rozdílné a kratší než na osobním počítači jako v našem případě.



Obrázek 16: Časy dekomprese vybraných sloupců pro 10 000 000 řádků.



Obrázek 17: Časy dekomprese vybraných sloupců pro 100 000 řádků.

Počet dotazů	1 000	10 000	100 000	1 000 000	10 000 000	100 000 000
Doba trvání [s]	0,004	0,041	0,341	4,344	40,129	425,221

Tabulka 13: Doby trvání vyhledávání v sekundách.

Obrázek 18: Časy vyhledávání v indexu.

6.5 Vyhledávání v indexu

V tomto dalším testu se podíváme na to jak dokážeme v komprimovaném indexu provádět operace, kde se ptáme na hodnotu na určité pozici v indexu. Operace pak probíhá tak, že máme zadanou pozici v bitmapovém indexu a v komprimovaném bitmapovém indexu musíme vyhledat zda se na této pozici nachází jednička nebo nula. Zde můžeme uvažovat, že doba vyhledání bude značně závislá na délkách sloupců indexů. Pokud budou sloupce dlouhé a budeme provádět dotazy na pozicích blízkých konci sloupců budou tyto doby značně odlišné od toho kdyby jsme dotazy prováděli na začátcích sloupců. To je zapříčiněno tím, že pokud se ptáme za pozice na konci indexu musí se zjistit všechny předem se vyskytující jedničky a to do té doby dokud nenalezneme jedničku na zvolené pozici nebo se dekomprimovaná jednička nachází již za námi hledanou pozicí.

Zde si uvedeme doby pro vyhledání v indexu s délkou sloupců 100 000 řádků. V tabulce 13 vidíme časy potřebné k danému počtu vyhledání v komprimovaném indexu. Můžeme jasně vidět, že se zvýšením počtu dotazů o jeden řád se adekvátně zvýší i potřebná doba. Tyto doby jsou relativně dlouhé a časově náročné. Z toho můžeme usoudit, že pro tento typ dotazů není komprimovaný index moc výhodný. Značně výhodnější je jej použít pro dotazy nad celými sloupci a operacemi s nimi. Pokud by jsme chtěli provádět dotazy na určité pozice v indexu bylo by lepší použít buď nekomprimovaný index a ptát se přímo na pozici nebo jinou formu indexu.

7 Závěr

I přes stále se zvětšující výpočetní výkon a úložný prostor dnešních počítačů, je stále kladen velký důraz na paměťové nároky zpracovávaných dat a jejich co možná nejrychlejší dostupnost. V případě databázových systémů se k urychlení vyhledávání žádaných dat využívá datová struktura zvaná index. Tato struktura, pokud je dobře použita, dokáže značně urychlit vykonávání databázových dotazů. Nicméně jako každá datová struktura i tato vyžaduje dodatečný paměťový prostor. Aby se paměťové nároky udržely v přijatelné míře, zavádí se komprimace indexů.

V naší bakalářské práci jsme se zabývali problematikou bitmapových indexů, jako jedním z druhů databázových indexů. Jejich nespornou výhodnou je jejich struktura, která umožňuje provádění velice rychlých bitových operací přímo nad jednotlivými částmi bitmapového indexu. Jako nevýhodu lze brát fakt, že nekomprimované bitmapové indexy se, se zvyšující kardinalitou indexovaného atributu stávají paměťově neúnosnými. Vysvětlili jsme si, jak vypadá struktura bitmapového indexu a jaké je jejich použití. Na příkladu byl také ilustrován postup zpracování databázového dotazu při použití bitmapových indexů. Po nastínění základní problematiky týkající se komprimace počítačových dat, se již práce zabývá konkrétně komprimací bitmapových indexů.

V části věnující se komprimaci je vysvětleno, proč nelze ke komprimaci použít pouze prosté binární reprezentaci délek sekvencí nul. Bylo představeno několik komprimačních schémat pro použití ke komprimaci bitmapových indexů, které byly srovnány. Vynecháno bylo pouze schéma RLE, které je popsáno v samostatné části jako komprimační schéma, jenž bylo vybráno pro účely této práce. Důvodem ke zvolení RLE jako hlavního komprimačního schématu, kterým se tato práce zabývala, byla snaha přiblížit základní principy komprimačních metod založených na run-length encodig, jelikož komprimační schémata jako BBC a WAH vycházejí z RLE. Dále byly vysvětleny postupy komprimace a dekomprimace, které je nutné dodržet pro správné provedení komprimace. To je důležité hlavně z důvodu, aby dekomprimace proběhla korektně a bylo nadále možné tento index používat, jak v dekomprimované podobě, tak v komprimované. Zároveň byla předvedena výhoda tohoto komprimačního schématu, které tkví v možnosti provádění databázových operací nad komprimovaným bitmapovým indexem. Na konkrétních příkladech byly také popsány jednotlivé postupy prováděné při komprimaci, dekomprimaci i operacích nad komprimovaným indexem. V době, kdy je vytvořen bitmapový index, je předpoklad, že data nad kterými je tento index vytvořen se nadále budou měnit. V tomto případě bylo nutné tedy uvést postupy jakými spravovat tento index. Ať už se jedná o přidávání nových záznamů do indexované tabulky, nebo jejich odebírání či modifikace. Důležité také je správné určení bitových vektorů indexu pro danou hodnotu atributu, aby bylo možné s tímto indexem pracovat.

Další z hlavních částí práce je samotná implementace komprimačního algoritmu, společně s možností dekomprimace. Další z nabízených funkcí jsou pak možnost dekomprimovat pouze určitý sloupec nebo sloupce, což může být vhodné zejména pro provádění operací nad celým sloupcem bitmapového indexu. Dále pak možnost pokládat dotazy na to zda se na určité pozici v indexu nachází jednička nebo nula. Jsou zde popsány důvody proč byly zvoleny dané datové struktury a také struktura samotných vstupních dat. Tyto

zvolené struktury jsou nedílnou součástí naší implementace a odvíjí se od nich úpravy a konkrétní podoba komprimačního algoritmu, který byl předtím teoreticky popsán. Byly popsány jednotlivé funkce programu a také důvody, proč bylo se stejným datovým typem nakládáno odlišně při komprimaci i dekomprimace. Tento důvod byl vztažen k omezení počtu zápisů jednotlivých bitů do datového typu použitého pro reprezentaci bitmapového indexu.

Poslední část práce se zabývá testováním našeho programu. Mezi testování bylo nutné zahrnou nejprve testy správnosti komprimace i dekomprimace, aby se zjistilo zda probíhá správně a není potřeba algoritmy patřičně upravit. Po zjištění správného průběhu bylo třeba určit komprimační poměry mezi vstupními daty a výstupními daty v podobě komprimovaného indexu. Takto jsme zjistili efektivitu naší implementace metody RLE. Bylo také provedeno porovnání dosažených výsledků a tyto výsledky zhodnoceny. Následovalo testování časové. Jelikož i kdybychom měli sebelepší komprimační algoritmus, který by ale neproběhl v konečném čase, byl by tento algoritmus takřka bezcenný. také jsme testovali rychlost dekomprimace v případě, že vybíráme pouze určitý počet nějakých sloupců v indexu. Na daných datech jsme zjistili kolik jsme schopni provést takovýchto dekomprimací za sekundu. Dalším z testovaných aspektů musíme zmínit rychlosti provádění dotazů na pozice v indexu. U této operace jsme zjistili, že provádění na komprimovaném indexu není zrovna efektivní a bylo by lepší použít buď nekomprimovaný index nebo index jiného druhu. Je také třeba brát v úvahu, že tyto časy se odvíjí od výkonu použitého počítače.

Ze zkušenosti s řešením této problematiky se dá usoudit, že bitmapové indexy v databázích mají své opodstatnění a se stále se zvětšující potřebou přístupu k datům najdou v budoucnu své uplatnění při indexování v databázích. Jakmile se tato možnost objeví u velkých hráčů na trhu databázových systémů. Jediným a hlavním omezením bitmapových indexů je jejich velká závislost na indexovaných datech, kdy již při malém zvětšení kardinality dat dochází k velkému nárůstu datového objemu indexu, za předpokladu, že počet záznamů je vysoký. V případě komprimace naše komprimační schéma dosahuje uspokojivých výsledků, nicméně bylo v práci zmíněno, že se dá také ještě vylepšit. Toto zlepšení vychází z vynechání prvního jedničkového bitu, čísla reprezentujícího nám délku kódované sekvence. Tato modifikace by byla vhodným kandidátem na navazující práci a bylo by jistě zajímavé porovnat výsledky těchto dvou metod. Námětem na další pokračování může být také implementace dalších dvou zmíněných komprimačních schémat (BBC, WAH) vycházejících z námi implementované metody RLE. V tomto případě by jistě bylo zajímavé porovnání, jak výkonnostních výsledků, tak velikosti komprimace a také pozorovat, zda-li by byla třeba obměna formátu vstupních dat, která jsme použili.

8 Reference

[1] Database index. In: Wikipedia: the free encyclopedia [online]. Aktualizováno 22. 4. 2014. [cit. 2014-04-25]. Dostupné z: http://en.wikipedia.org/wiki/Database_index

[2] Bitmap index. In: Wikipedia: the free encyclopedia [online]. Aktualizováno 27. 9. 2013. [cit. 2013-10-13]. Dostupné z: http://en.wikipedia.org/wiki/Bitmap_index

[3] O'Neil, E.; O'Neil, P.; Kesheng Wu, "Bitmap Index Design Choices and Their Performance Implications,"*Database Engineering and Applications Symposium*, 2007. IDEAS 2007. 11th International , vol., no., pp.72,84, 6-8 Sept. 2007.

[4] LIU, Ling a M ÖZSU. Bitmap index. In: *Encyclopedia of database systems*. New York: Springer,2009, 3749 str. ISBN 978-038-7496-160.

[5] LIU, Ling a M ÖZSU. Indexing of Data Warehouses In: *Encyclopedia of database systems*. New York: Springer,2009, 3749 str. ISBN 978-038-7496-160.

[6] Data compression. In: Wikipedia: the free encyclopedia [online]. Aktualizováno 9. 1. 2014 [cit. 2014-01-24]. Dostupné z: http://en.wikipedia.org/wiki/Data_compression

[7] KRČMÁŘ, Petr. Unixová komprese v praxi: Úvod. ROOT.CZ [online]. Aktualizováno 8. 3. 2003 [cit. 2013-11-5]. Dostupné z: http://www.root.cz/clanky/unixova-komprese-v-praxi-uvod/

[8] DIGITAL EQUIPMENT CORPORATION. *Byte aligned data compression* [online]. Vynálezce: Gennady Antoshenkov.. Přihl. 25.11.1994. MPT: G06T 9/00; H03M 7/46; H03M 007/00 ; H03M 007/46. Čís. patentu US5363098 A. The United States Patent and Trademark Office. [vid. 2.7.2014]. Dostupné na: http://patft.uspto.gov/netacgi/nph-Parser?Sect2=PTO1&Sect2=HITOFF&p=1&u=/netahtml/PTO/search-bool.html&r=1&f=G&l=50&d=PALL&RefSrch=yes&Query=PN/5363098

[9] THE REGENTS OF UNIVERSITY OF CALIFORNIA. *Word aligned bitmap compression method, data structure, and apparatus*[online]. Vynálezce: Kesheng Wu, Arie Shoshani, Ekow Otoo. Přihl. 14.12.2004. MPT: G06T 9/00; H03M 7/30; H03M 007/00. Čís. patentu US6831575 B2. The United States Patent and Trademark Office. [vid. 2.7.2014]. Dostupné na: http://patft.uspto.gov/netacgi/nph-Parser?Sect2=PTO1&Sect2=HITOFF&p=1&u=/netahtml/PTO/search-bool.html&r=1&f=G&l=50&d=PALL&RefSrch=yes&Query=PN/6831575

[10] WU, Kesheng; OTOO, Ekow J.; SHOSHANI, Arie. Optimizing bitmap indices with efficient compression. *ACM Transactions on Database Systems (TODS)*, 2006, 31.1: 1-38.

[11] ULLMAN, Jeffrey D.; GARCIA-MOLINA, Hector; WIDOM, Jennifer. *Database Systems: The Complete Book*. New Jersey: Prentice Hall, 2002, 1119str. ISBN 0-13-031995-3.

[12] A Brief Description. [online]. [cit. 2014-03-01]. Dostupné z: http://www.cplusplus.com/info/description/

[13] History of C++. [online]. [cit. 2014-03-01]. Dostupné z: http://www.cplusplus.com/info/history/

[14] SALOMON, David. *Data Compression: the complete reference*. 4th ed. London: Springer, 2007, s. 23-31. ISBN 978-1-84628-602-5.

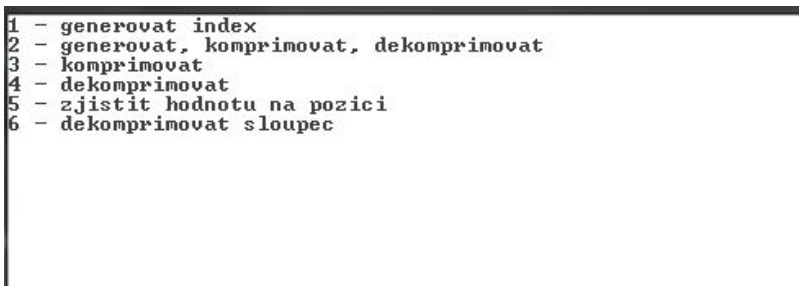
[15] STL Containers. [online]. [cit. 2014-03-10]. Dostupné z: http://msdn.microsoft.com/en-us/library/1fe2x6kt.aspx

[16] Std::vector. [online]. [cit. 2014-03-10]. Dostupné z: http://www.cplusplus.com/reference/vector/vector/

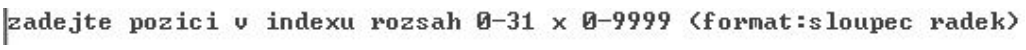
[17] Std::vector<bool>. [online]. [cit. 2014-03-10]. Dostupné z: http://www.cplusplus.com/reference/vector/vector-bool/

A Návod k programu

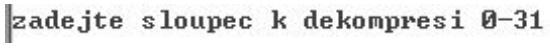
Při spuštění programu máme na výběr z několika možností. Jak můžeme vidět na obrázku 19. První možnost slouží k vytvoření nekomprimovaného indexu o zadané velikosti. Druhá slouží k provedené všech operací a tedy vytvoření indexu, jeho komprimaci i dekomprimaci. Další dvě možnosti slouží pro komprimaci nebo dekomprimaci. Pátá možnost slouží k zjištění hodnoty indexu na dané pozici. Uživatel zadá pozici k zjištění viz. obrázek 20. Poslední možností je dekomprimovat zadaný sloupec. Opět je uživatel vyzván k volbě sloupce viz. obrázek 21.Veškerá volby je po zadání příslušného čísla nutné potvrdit stisknutím klávesy enter.



Obrázek 19: Hlavní nabídka.



Obrázek 20: Výběr pozice v indexu.



Obrázek 21: Výběr sloupce k dekomprimaci.